Computer Science

A Language for Higher-Order Explanation-Based Learning

Scott Dietzen

January 1992
CMU-CS-92-110

Carnegie Mellon

A Language for Higher-Order Explanation-Based Learning

Scott Dietzen

January 1992
CMU-CS-92-110

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

A Language for Higher-Order Explanation-Based Learning

SCOTT DIETZEN

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Frank Pfenning — MAJOR PROFESSOR — 1/27/92 — DATE

[signature] — MAJOR PROFESSOR — 1/27/92 — DATE

[signature] — DEAN — 3/13/92 — DATE

APPROVED:

[signature] — PROVOST — 3/18/92 — DATE

Abstract

Certain tasks, such as formal program development and theorem proving, fundamentally rely upon the manipulation of *higher-order* objects such as functions and predicates. Computing tools intended to assist in performing these tasks are at present inadequate in both the amount of 'knowledge' they contain (*i.e.*, the level of support they provide) and in their ability to 'learn' (*i.e.*, their capacity to enhance that support over time). The application of a relevant machine learning technique — *explanation-based generalization (EBG)* — has thus far been limited to first-order problem representations. We extend EBG to generalize higher-order values, thereby enabling its application to higher-order problem encodings.

Logic programming provides a uniform framework in which all aspects of explanation-based generalization and learning may be defined and carried out. First-order Horn logics (*e.g.*, Prolog) are not, however, well suited to higher-order applications. Instead, we employ *λProlog*, a higher-order logic programming language, as our basic framework for realizing higher-order EBG. In order to capture the distinction between *domain theory* and *training instance* upon which EBG relies, we extend λProlog with the necessity operator □ of *modal logic*. We then provide a formal characterization of both the extended logic, $\lambda^{\Box}$*Prolog*, and of higher-order EBG over $\lambda^{\Box}$Prolog computation. We also illustrate applications of higher-order EBG within program development and theorem proving.

Within the architectures of traditional learning systems, the language for problem representation and solution (*i.e.*, the programming language) is separated from the underlying learning mechanism. Herein we propose an alternative paradigm in which generalization and assimilation are realized through integrated features of the programming language, and are therefore under programmer control. In this way, the developer can leverage domain knowledge and provision for user interaction in the programming of learning tasks. Thus, while $\lambda^{\Box}_{\mathcal{G}}$Prolog — the logic extended with generalization and assimilation features — is not itself a learning system, it is intended to serve as a flexible, high-level foundation for the construction of such systems.

For $\lambda^{\Box}_{\mathcal{G}}$Prolog to afford this programmable learning, constructs are necessary for controlling generalization, and for assimilating the results of generalization within the logic program. The problem with the standard means by which Prolog programs are extended — **assert** — is that the construct is not *semantically well-behaved*. A more elegant alternative (adopted, for example, in λProlog) is implication with its *intuitionistic* meaning, but the assumptions so added to a logic program are of limited applicability. We propose a new construct **rule**, which combines the *declarative semantics* of implication with some of the power of **assert**. *Operationally*, **rule** provides for the extension of the logic program with results that deductively follow from that program. We then extend **rule** to address explanation-based generalization within another new construct, **rule_ebg**. While **rule** and **rule_ebg** are developed in the framework of λProlog, the underlying ideas are general, and therefore applicable to other logic programming languages.

In addition to developing and formally characterizing the $\lambda^{\Box}_{\mathcal{G}}$Prolog language, this thesis also provides a prototype implementation and numerous examples.

Thesis Committee: **Frank Pfenning**, Thesis Advisor
Carnegie Mellon University

William Scherlis, Advisor
Carnegie Mellon University & DARPA/ISTO

Dale Miller
University of Pennsylvania

Tom Mitchell
Carnegie Mellon University

Author's Present Address: Scott Dietzen
Transarc Corporation
707 Grant Street
Pittsburgh, Pennsylvania 15219
USA
dietzen@transarc.com

Contents

Chapter 1

Introduction

Broadly speaking, this thesis should be viewed as a language design effort. Rather than starting from scratch, this effort considers extensions to the higher-order logic programming language λProlog [100], itself the subject of current research. These enhancements focus upon the incorporation of generalization and learning, and in particular, explanation-based generalization (EBG) and learning (EBL), within the framework of λProlog. While learning is central to this work, the dissertation does not contain any demonstrations of performance improvement, such as through timing evaluations or learning curves. This is because the focus of this thesis is not a 'stand alone' problem solver that learns. Rather, it is a programming language — $\lambda^{\Box}_{\mathcal{G}}$Prolog.

The preceding distinction is fundamentally important to the evaluation of this work: First, unlike typical learning systems, $\lambda^{\Box}_{\mathcal{G}}$Prolog does not pose its own learning problems. Instead, $\lambda^{\Box}_{\mathcal{G}}$Prolog incorporates constructs that provide for *programmable* generalization and assimilation. By integrating learning mechanisms within the programming language, we defer one of the more difficult problems faced by a 'learner': determining over what computations to attempt learning, or in other words, determining when to learn. Our approach allows the programmer (or *client*) to explicitly control learning within the same language as that in which the problem is encoded. We claim that it is the client which can best coordinate learning, as he is in the best position to leverage domain knowledge and user-interaction. Although $\lambda^{\Box}_{\mathcal{G}}$Prolog is not itself a learning system, it is intended to serve as a high-level foundation for the implementation of such systems.

This thesis, however, embodies more than just a novel approach to the formulation of learning tasks within logic programs: we herein extend and reformulate the paradigm of EBG, and moreover, develop semantically straightforward means by which EBG and more limited forms of generalization can be integrated within the logic programming framework. Some background is in order.

Higher-order representation languages. We distinguish the *representation language* to be the subset of a programming language concerned with the encoding (or representation) of data. For conventional programming languages, the representation language includes expressions for booleans, integers, reals, strings, *etc.* Typical representation languages are

first-order in that there are no special primitives to encode *higher-order*, or argument taking values, such as functions, procedures, and predicates. This, in turn, means that in order to program over higher-order domains (that is, formulate tasks that manipulate higher-order objects), the client must himself come up with an encoding, and then explicitly program fundamental higher-order operations (*e.g.*, substitution, matching, the occurs check[1]). To some extent, this situation is analogous to a programmer having to explicitly code basic arithmetic or string operations.

Of course, for the vast majority of programming, the data being represented *is* first-order, and thus higher-order representation languages are irrelevant. However, for programming over higher-order domains — for example, mathematics (where the objects to be manipulated include functions and predicates), and programming itself (in which programs are manipulated) — higher-order expressivity is of substantial importance. Witness the success of higher-order programming languages such as ML, LISP, Scheme, and λProlog.

Explanation-based generalization. We have mentioned that higher-order languages are particularly suited to such tasks as formal program development and theorem proving. The tools which perform or assist such tasks, however, are at present inadequate in both the amount of 'knowledge' they contain (*i.e.*, the level of support they provide) and in their ability to 'learn' (*i.e.*, their capacity to enhance that support over time).

The application of a relevant machine learning technique, explanation-based generalization, has thus far been limited to generalizing first-order representation languages. By extending this technique to *higher-order* EBG — explanation-based generalization in which the candidates for generalization include higher-order objects, we facilitate EBG's application to such naturally higher-order domains as program development and theorem proving.

EBG establishes the weakest preconditions sufficient to apply a particular problem solving strategy in general, thereby speeding the subsequent solution of analogous problems. Recently, the logic programming paradigm has been touted as a foundation for EBG, because of its declarative nature, due to its support for unification and search, and because it admits a common representation for all aspects of EBG. However, for the domains in which we are interested, the first-order representation language of Prolog is inadequate. *Instead, we provide a formulation of higher-order EBG over λProlog — one of the fundamental contributions of this dissertation.*

'Logical' and programmable mechanisms for controlling generalization and assimilation. In order to utilize these generalizations, there must be a mechanism for extending the existing logic program $\mathcal{P}$ with new clauses. The problem with the standard means by which Prolog programs are extended — **assert** — is that the construct is not semantically well-behaved. As a result, programs making use of **assert** are harder to reason about and manipulate (*e.g.*, compile). For this and other reasons, **assert** is not part of

[1]The occurs check, as familiar from logic programming, determines whether a variable occurs free within an expression [124, pp.69–70].

λProlog. We herein propose a new construct, **rule**, which has some of the power of **assert**, but also offers a straightforward semantics reconcilable with λProlog. Operationally, **rule** provides for the extension of the logic program with results that deductively follow from that program. Later we devise an analogous construct, **rule_ebg**, that constructs and assimilates explanation-based generalizations. *Herein lies a second thesis contribution: semantically straightforward constructs for controlling generalization and assimilation within the logic programming framework.*

$\lambda^{\Box}_{\mathcal{G}}$Prolog. The enhancement of λProlog with higher-order EBG and with a means for coordinating learning yields $\lambda^{\Box}_{\mathcal{G}}$Prolog. The early chapters of this dissertation set the stage for this language by developing each of the preceding topics, relying heavily throughout upon examples. (In fact, our examples where produced via a prototype implementation of $\lambda^{\Box}_{\mathcal{G}}$Prolog.) *The third thesis contribution is this language itself, which we claim is an attractive vehicle for the formulation of tasks that benefit from its higher-order representation logic and programmable generalization and learning.* Moreover, $\lambda^{\Box}_{\mathcal{G}}$Prolog provides this functionality in such a way that user-interaction (for the addressing of problems beyond the capabilities of the logic program itself) can be smoothly integrated.

1.1 Thesis Contributions

In somewhat more concrete terms, this thesis contributes the following:

- The **rule** construct, which provides a semantically sound means for *universal generalization* (*i.e.*, the selective universal quantification of free variables) within the logic programming paradigm, and in particular λProlog.
- A formal account of **rule**, and its variant **lemma**.
- An alternative formulation of the EBG paradigm relying upon modal logic to formalize the heretofore tacit distinction between domain and training theory.
- The extension of the EBG algorithm to treat higher-order representation language.
- A formal account of higher-order EBG (in terms of a λProlog meta-interpreter).
- The realization of EBG as a programmable feature of $\lambda^{\Box}_{\mathcal{G}}$Prolog through a generalization of the **rule** construct — **rule_ebg**.
- A formal account of **rule_ebg**, and its variant **lemma_ebg**.
- The integration of all of the above within $\lambda^{\Box}_{\mathcal{G}}$Prolog in a manner that admits user-guided problem solving and generalization.
- Numerous examples.

1.2 Acknowledgments

The research embodied within this dissertation is extensively collaborative with my thesis advisor Frank Pfenning. Frank initiated the effort, and has participated in virtually every aspect since. It would be difficult to overstate his contribution, at least short of granting the author a Ph.D. Without his insight, his enthusiasm, and his willingness to sweat the details with me, this work would not have come to fruition.

Indicative of these collaborations, I typically use 'we' rather than 'I' within my own prose,[2] and I have also taken the occasional liberty of borrowing and adapting passages from previous joint work [32, 31].

The contribution of my co-advisor, William Scherlis, is of a different sort: Bill has provided me with knowledge and guidance since my days as an undergraduate. And while he has not directly participated in this research, he has made a substantial effort helping me to better present the ideas herein, as well as helping me to establish the framework in which to cast this dissertation. Symptomatic of this, I have also taken the liberty of incorporating a couple of passages from Dietzen & Scherlis [33] within Chapter 2.

The remaining members of my committee — Dale Miller and Tom Mitchell — have generously provided their time and expertise toward the development and articulation of the ideas embodied herein. Their participation is particularly important because of their respective contributions to two themes centrally melded within this dissertation: higher-order logic programming and explanation-based learning.

I would further like to thank other individuals providing thoughtful discussions and comments either directly pertaining to this work or to the presentation thereof: Penny Anderson, Conal Elliott, Masami Hagiya, Bob Harper, Nevin Heintze, Haym Hirsh, Spiro Michaylov, Jack Mostow, and several anonymous referees.

Finally, I gratefully recognize the knowledge, encouragement, and last but not least, the support I have derived over the years from my association with the faculty, students and staff of the Ergo Project and its parent, Carnegie Mellon's School of Computer Science.

[2]Nevertheless, the author bares full responsibility for any errors or omissions within this presentation.

Chapter 2

Motivation

This chapter attempts to establish an appropriate framework for the ideas developed within this dissertation. To that end, it is necessary to discuss the application of *formal methods* to program development and theorem proving — a topic much more thoroughly and eloquently treated by others: see, for example, Dijkstra [34, 36], Sintzoff [119], Gries [51], Broy [11], Bauer [5], Scherlis & Scott [116], Balzer, Cheatham & Green [2], Meyer [82], Constable, *et al.* [20], Barwise [3], and more recently, Gerhart [48] and Wing [135]. Readers primarily interested in our extensions of the explanation-based learning paradigm or in our enhancements to the logic programming language λProlog, may find this chapter largely superfluous.

The results of the programming process, namely programs, are necessarily expressed *formally*, that is, within *formal language*. Similarly, an increasing amount of mathematics is carried out within formal language. A formal language is one that is mathematically precise — that is, it has a well-defined syntax (*e.g.*, through a BNF grammar) and a well-defined semantics (*e.g.*, through a mathematical model). *Formalization* is, then, the process of codifying ideas expressed *informally* (*e.g.*, in a natural language) within a formal language. In general, formalization requires resolving ambiguity, thereby achieving the precise expression (and hence communication) of concepts. The classification 'formal method' is a term applied to paradigms that more strongly rely upon formal language.

While there exist a wealth of tools to assist the tasks of program development and theorem proving, the level of support they provide is generally inadequate. If we are to build tools that offer a substantially higher level of functionality, it is essential that we continue to place a greater reliance upon formal methodologies. This is due to the fact that formal techniques facilitate a wider and deeper penetration of machine support, simply because they require that more of the relevant 'knowledge' be encoded in a formal (*i.e.*, a machine manipulable) language.

Of course, successfully coding programming or theorem proving 'knowledge' *a priori* is impossible due to the scope, complexity, and evolutionary nature of these domains. Rather, tools must support the *assimilation* of experience gained in the course of solving problems. However, simply memoizing (*i.e.*, caching) particular solutions will be insufficient; instead,

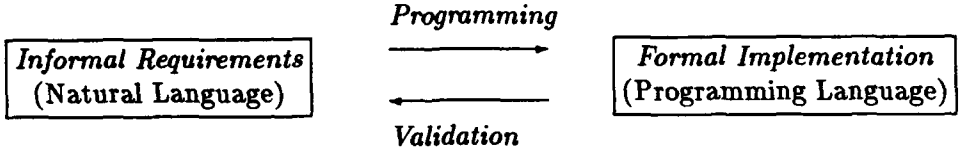

Figure 2.1: Initial Approach

experience must be *abstracted* or *generalized*. *Learning*, the ability to generalize and assimilate from experience, will therefore have a significant impact on the success of future tools and methods.

The vehicle we have chosen for experimenting with generalization and learning over the domains of program development and theorem proving is the higher-order logic programming language λProlog. These domains demand a higher-order treatment in that they require the manipulation of higher-order objects, such as functions and predicates. λProlog provides the simply-typed λ-calculus for the representation of higher-order objects, and furthermore supports higher-order programming — that is, the ability to create goals and programs and pass them as arguments. The latter concern is particularly relevant for the realization of programming tools, since it affords the ability to manipulate the logic program itself.

2.1 New Computing Tools for Programming and Mathematics

2.1.1 Tools for Programming

An initial view of the programming process is depicted in Figure 2.1: given a set of informal requirements, programming is the task of constructing a (formal) program to meet those needs. This paradigm is, however, problematic. Consider that programmers are often given the task of determining what it is that existing code does. Can you answer that question for the program of Figure 2.2, which is taken from Bentley's *Writing Efficient Programs* [6, p.60]?[1] The only substantive modification I have made is to replace the procedure's descriptive name with *f*. The answer is given below.

Program specification and program abstraction. One difficulty with producing understandable code is that the goals of clarity and efficiency tend to be mutually exclusive. Now consider a *formal specification* of the preceding program, given in Figure 2.4. From this, the reader presumably has little trouble recognizing the Fibonacci function. What is

[1]The choice of example is borrowed from a presentation by William Scherlis.

```
f(n) ⇐ var a,b,i : integer
       begin
         if n ≤ 0 then return 0;
         if n ≤ 2 then return 1;
         a ← 1;
         b ← 1;
         for i ← 1 to (n div 2) do begin
           a ← a + b;
           b ← b + a
           end;
         if odd(n) then b ← b + a;
         return b
       end
```

Figure 2.2: A puzzling program

---

it that distinguishes specification and implementation? Primarily, the specification is more *abstract*, in that much of the detail of the implementation has been omitted, while the implementation is more *efficient* (in this case because it does not recompute values). In general, programs are made efficient by making commitments to data representation, order of computation, *etc.*, and then optimizing on the basis of those commitments. This process of *specialization* necessarily complicates the functionality with procedural detail capturing how that functionality is to be achieved [115, 116].

Like the implementation, the Fibonacci specification is *formal* since it is expressed in a (potentially) formal language. Languages for formal specification are generally characterized as *abstract*, *very high-level*, or *wide-spectrum*, and often are nondeterministic or nonexecutable. The term 'wide-spectrum' is indicative of the same language serving for both specification and implementation (although either might be restricted to a particular subset of that language).

The construction of a formal specification divides the task of programming into two parts: the transition from informal requirements to formal specification, or 'what', and the transition from specification to efficient implementation, or 'how.' In the ideal, then, specification languages serve to express the program's intended functionality unencumbered by details of computation strategy. (Our use of the term 'program' is intended to encompass the spectrum from specification to implementation.) The resulting programming methodology that results is illustrated in Figure 2.3. A substantial number of general and special purpose specification languages have been developed: consider, just for example, Larch [52, 53], CIP–L [4], GIST [134], Refine [120], Z [122], and RAISE [101]. (Meyer provides an accessible introduction to formal specification and contrasts the approach with natural language specification [82], while Wing gives an overview [135].)

Our discussion of high-level languages has not touched upon the range of useful abstrac-

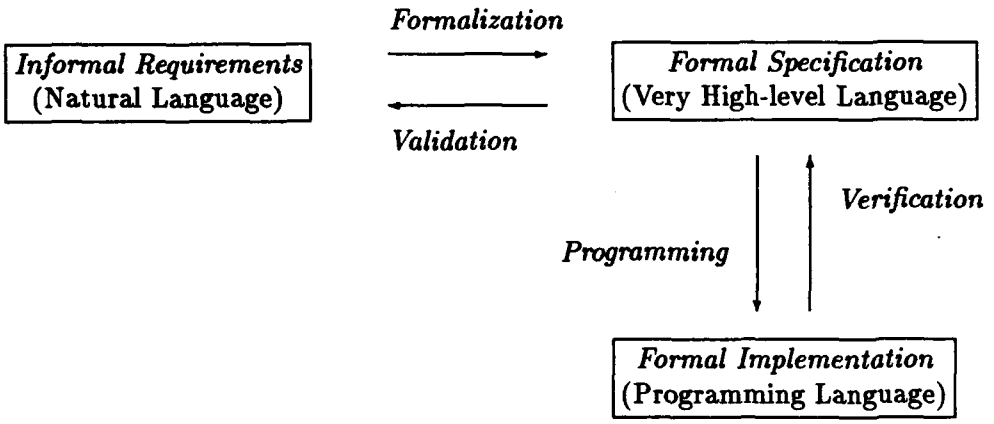

Figure 2.3: Program Specification Approach

$$
\begin{aligned}
f(0) &= 1 \\
f(1) &= 1 \\
f(n) &= f(n-1) + f(n-2)
\end{aligned}
$$

Figure 2.4: A less puzzling program

tion techniques in programming languages: Consider, again just for example, abstract data types, polymorphism, program libraries, object-oriented techniques, Dershowitz's program templates [28], idioms in the Programmer's Apprentice [111, 133], structuring techniques such as module and interface definitions, and finally those abstractions associated with computer-aided software engineering (CASE) management tools [18]. But while abstractions of programming language go a long way toward easing the task of program development, we claim that they are inherently limited.

Program rationales. Consider again the task of understanding the Fibonacci implementation of Figure 2.2. Given that the formal specification captures what the implementation is doing, the programmer is still left to determine how it is accomplished. *Reverse engineering* refers to this problem of *a posteriori* reconstructing the *rationale* for an implementation, a nontrivial task for even our simple example. An informal rationale linking the Fibonacci specification and implementation is given in Figure 2.5. We claim that it is the combination of specification and rationale that best elucidate the implementation.

What should be the real results of the program design process? If program *executions* were the sole results of interest, then it would be sufficient that the result of programming be simply a *program.* But this is rarely the case in practice. Most programs undergo analysis, modification, and adaptation, usually during their original development. Under these circumstances, delivering the program itself is simply not sufficient. An indication is the

1. Compute successive pairs of Fibonacci numbers with a tail recursive program:

$$\langle u, v \rangle \leftarrow \langle v, u + v \rangle$$

2. Replace tail-recursion with iteration.

3. Unroll the loop once. This allows the removal of a trivial assignment and a temporary variable.

Figure 2.5: Fibonacci Rationale

---

experience of software maintainers, who spend the majority of their time reverse engineering existing code. (Lientz & Swanson's survey claims that maintenance represents approximately 70% of software cost [78].) The problem is that more knowledge has been brought to bear on the implementation of a program than is evident in the code alone. This knowledge may be presented in the form of a rationale or *design history* of the program [116, 33]. Within a design-based paradigm, the designed objects should not be programs, but rather program rationales.

Program modification is ordinarily very difficult for programmers because, like *a posteriori* verification, it requires rediscovery of concepts used during the development of the implementation. But by preserving the rationale of the initial program, it is often possible to pinpoint the design decision that must be altered, and carry over (*i.e.*, *replay* [97]) much of the remaining structure of the original development.

Figure 2.6 illustrates this paradigm. The new formal object — the design record — serves as a 'road map' from specification to implementation; that is, it captures the sequence of design decisions (represented formally) from which the implementation can be derived from the specification. The rationale may be considered a *meta-program* in that it is a program that manipulates other programs.

Successive refinement paradigms are a step toward design-based development in that the informal programming process is replaced with a series of programs, beginning with the specification and leading to the implementation, each of which is generally of better performance than its predecessor. Consider, for example, capturing the evolution of Fibonacci with a series of programs, each annotated with the appropriate comment from the rationale of Figure 2.5. These successive programs might be expressed within a single wide-spectrum language or within several layers of increasingly concrete languages. The successive refinement model divides program development into more intelligible and more easily justified steps, in that way affording greater confidence in the resulting implementation.

There are, however, several reasons to encode program rationales formally rather than informally. First, by making design records formal objects, we increase the likelihood that they will actually be written and maintained. Consider the cynical yet all too relevant words of

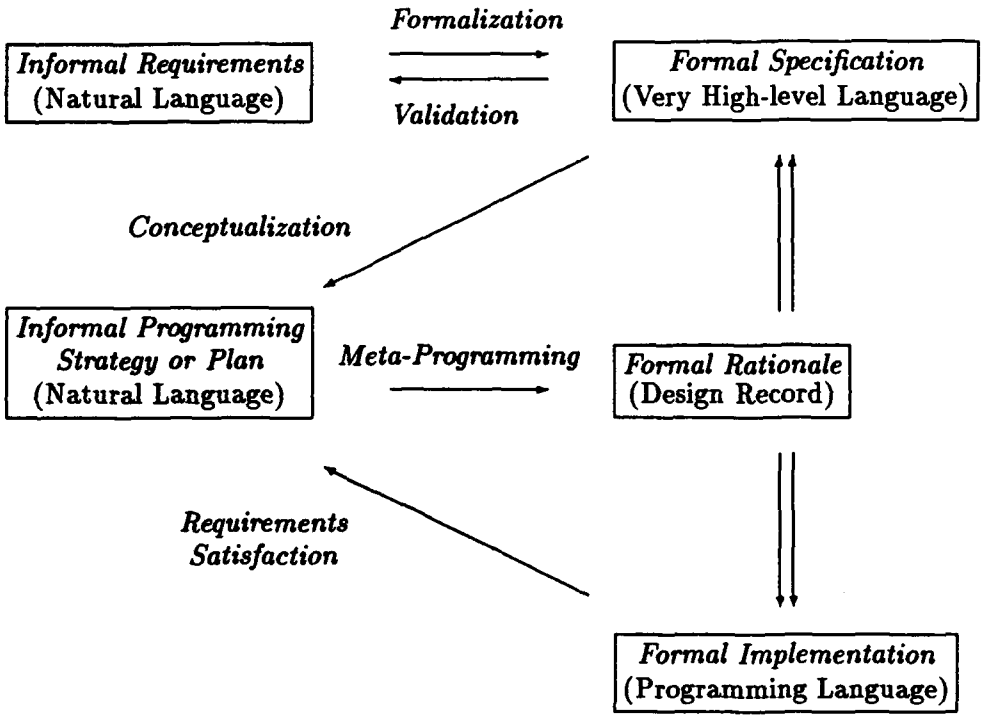

Figure 2.6: Program Rationale Approach

Dave Storer [7, p.60]: "Don't get suckered in by the comments — they can be terribly misleading. Debug only the code." More significantly, a formal design history is by definition precise and, further, allows us to make claims about the relationship between specification and implementation. For example, if each of the design steps is *truth preserving*, the resulting implementation is *correct by construction.* Finally, explicit formal rationales may themselves be manipulated by computing tools: in particular, they may be abstracted (to produce programming strategy) and *reused* on related problems.

The design-based paradigm of programming is by no means a new idea: it has been espoused by many researchers through a series of methodologies over several years. These approaches range from unstructured successive prototyping schemes to design frameworks in which an *account* (a rationale) is associated with the individual steps of program development. Within these structured design-based paradigms, rationales may themselves be formal (rules, proof steps) or informal (comments). The following enumerates work on the formal end of this spectrum:

- *Program transformation* (*e.g.*, [13, 42, 68, 115, 121]) — This approach is best visualized as a state/operator space in which the states are programs and the operations are transformations mapping between programs. Typically, program transformations are largely syntactic manipulations, which improve efficiency when sequenced effectively. They may or may not preserve the meaning of the program. Transformations have been applied in a number of different ways:

  - automatically, often within the compilers of high-level languages. Here no explicit design record is built; rather it is implicit in the transformation engine or expressed by directives given to the compiler (typically as annotations within the programming language).
  - via an explicit meta-program — a sequence of transformations specified *a priori.*
  - interactively.

  The term 'program transformation' is very broad, encompassing any well-defined operation on a program. (See Chapter 7 for examples and further discussion.)

- *Interactive constructive programming* or *proofs as programs* (*e.g.*, NuPRL [20], Calculus of Constructions [21]) — These are paradigms in which the programmer proves a theorem (or specification) *constructively*, thereby implicitly defining an algorithm for producing the answer. The program is then *extracted* from the proof.

- *Proof transformation* [104, 80] — This is a hybrid of the transformation and proof-based approaches in which proofs rather than programs are transformed. This combination moves correctness considerations outside of the transformation semantics, since the resulting proofs may be checked for validity (presumably by machine).

2.1.2 Tools for Mathematics

Unlike programs, mathematical proofs are not inherently formal: their role has simply been to convince other mathematicians of a theorems validity [24, 3]. And in terms of human consumption, informal arguments are generally superior to formal ones. Nevertheless, in order to enhance the application of computing tools to theorem proving tasks, proofs should be represented formally. The same arguments applied within the discussion of program development are relevant here: (1) formal proofs are precise, (2) they may be manipulated, and in particular, checked and/or reapplied by a machine, and (3) they may be generalized (*i.e.*, abstracted) by machine to represent theorem proving strategy, and thereby utilized in new situations.

The above discussion suggests an analogy between theorems (mathematical formulas) and specifications (programs), and also between proofs and program rationales [116]. We shall use the term *derivation* to encompass the formal reasoning supporting either a theorem or a program implementation, and the discussion to follow pertains to both domains. In fact, for the 'proofs as programs' paradigm, the process of programming is reduced to one of theorem proving.

2.1.3 Formalization

Formal methods have been maligned in the literature: see in particular, De Millo, Lipton & Perlis [24] and Fetzer [45]. The essential issue is the open philosophical question of the extent to which mathematical and programming knowledge *can* be formalized. To some degree, the deeper relevance of the thesis rests upon the formalist's viewpoint: that any line of reasoning may be formalized. This discussion is, however, beyond the scope of this thesis. Interested readers should pursue the above references, as well as Barwise's reply [3] and that offered by Scherlis and Scott [116]. The debate has continued most recently with an article by Dijkstra on computer science education and the responses it elicited [35].

A more limited criticism of formal methods, espoused for example by some software engineers, is that while formal techniques work well for mathematically elegant problems (*e.g.*, Fibonacci), they are ineffective at addressing the range and scale of problems faced by programmers. Our response is fourfold: First, we need more expressive and special purpose specification languages that can formally, concisely, and elegantly capture the functionality of a greater variety of systems. Second, we need more expressive meta-languages to capture programming design decisions, again formally, concisely, and elegantly. Third, the tools supporting formal methodologies must provide substantially greater levels of assistance to the user. Of course, the goals of better languages and better tools are *not* mutually independent: progress on tools is limited by the elegance and expressivity of the language, and language improvements will come out of better tools.

And finally, it must be recognized that formal methods are one component of an overall software engineering strategy: In suggesting the design-based view, we are *not* advocating the replacement of existing methodologies and tools with some new all-encompassing paradigm. Instead, we believe that for certain tasks — for example, highly optimized or mathematical

algorithms, or procedures requiring a high degree of confidence — the design-based approach will become feasible. The way to proceed, then, is to make design-based paradigms more broadly applicable, to make them easier to use, and to integrate them within existing methodologies and tools.

2.2 The Relevance of Logic Programming and λProlog

Generally, search and unification (matching) are integral components of theorem provers and formal programming tools. Logic programming languages are particularly suited to these domains because of their implicit support for both search and unification. This allows the programmer to focus at a high-level, on the 'logic' if you will, with less regard for underlying implementation details. The result is a more clear and concise formulation of many problem domains.

At the same time, the tasks of program development and theorem proving are fundamentally *higher-order*, since they rely upon the manipulation of higher-order objects — objects that take other objects as arguments. Higher-order objects are most naturally represented with a name binding operator such as λ. λProlog provides for the elegant representation of higher-order objects in that it contains the typed λ-calculus as a datatype. And furthermore, λProlog affords the manipulation of these higher-order encodings by providing the λ-specific operations of $\alpha\beta\eta$-conversion as well as higher-order unification.

λProlog is also an attractive *meta-language* for the expression of formal rationales and proofs. This is because of the above, and because λProlog offers the further expressivity of nested implication and explicit quantification. The language also provides a degree of polymorphism, which allows logic programs (and thus derivations) to be abstracted over a particular datatype.

In addition to higher-order objects, λProlog supports *higher-order programming*, in that one may create goals and programs and pass them as arguments. This allows us to naturally reason at the meta-level, that is about the logic program itself. The expressiveness of higher-order programming becomes important with multiple meta-layers of language, because it elegantly facilitates *reflection* — the mapping from data to executable logic program, and *reification* — the reverse mapping (§5.3.1). Indeed, since we are interested in program development paradigms and theorem provers, we further use λProlog (albeit in a limited way) as a *meta-meta-language*, for expressing the manipulation of derivations (*i.e.*, rationales and proofs).

In summary, λProlog is an attractive framework in which to experiment with tools for theorem proving and program development because it is both a *logic* programming language and a *higher-order* programming language. We expand upon this discussion within the next chapter. A more complete argument is made within Felty & Miller [43] and Hannan & Miller [59].

2.3 The Need for Generalization and Learning

To generalize an expression is to make that expression less *specific* — that is, to make it, in some sense, more broadly applicable. *To abstract* an expression, on the other hand, is to make it less *detailed* — that is, to abbreviate it in some manner. Typically, abstraction and generalization coincide: operations that make an expression more broadly applicable most often remove detail: for example, replacing a constant in **rained tuesday** with a variable in **rained** x.[2] For the discussion to follow, we use the terms generalization and abstraction interchangeably.

In the preceding sections, we suggested formal representation of derivations as a means by which to capture problem solving experience or design knowledge. An overriding limitation of formal derivations is that they are often so verbose as to be unintelligible. Indeed, the attraction of informal rationales is that they omit less pertinent details. Generalization is thus a potential avenue for making formal methods more palatable to the user. For example, it is our belief that by abstracting sequences of low-level derivation steps, we can produce more lucid, high-level derivations (much as high-level programming languages abstract sequences of machine instructions).

Effective tools for supporting formal methods will contain a substantial store of information, such as general programming techniques, previously solved problems (e.g., objects and their associated methods), problem domain theories, or derived mathematical results. As mentioned earlier, successfully coding this 'knowledge' *a priori* is impossible due to its scope, complexity, and evolutionary nature. Rather, tools must support the *assimilation* of experience gained in the course of solving problems. Our previous claim — that formal tools must support *learning* — simply means that they must facilitate the growth of this knowledge base, and that such growth must often include abstracted or generalized experience. The ultimate goal for generalization and learning within a designed-based paradigm is to enable the construction of libraries of derivations and abstracted derivations analogous to (and used in combination with) the program libraries of today.

We do not suggest that the technique of higher-order explanation-based generalization and learning developed herein is sufficient for automating the spectrum of generalization techniques necessary to realize this vision. Rather, we believe that EBL is an attractive technique to explore, particularly because validity is not sacrificed in the generalization process: that is, the results of explanation-based generalization are guaranteed to be sound (*i.e.*, follow from the existing theory).

---

[2]One notable exception to this is disjunction: while **rained tuesday** $\vee$ **rained wednesday** is more 'general' (*i.e.*, less specific) than **rained tuesday**, it is not more 'abstract' (*i.e.*, less detailed).

Chapter 3

λProlog — A Higher-Order Logic Programming Language

Nadathur & Miller introduce the higher-order logic programming language λProlog [100]. λProlog extends traditional logic programming languages

- by providing the simply-typed λ-calculus as a data-type,
- by incorporating the higher-order unification required for λ-terms,
- by including more expressive logic constructs: embedded implication and explicit quantification,
- by admitting higher-order predicates in a principled manner,
- by providing a degree of polymorphism, and
- by supporting abstraction mechanisms such as modules and higher-order data-types.

Within this chapter we briefly introduce λProlog. While this work relies upon and extends λProlog, the language is itself a research prototype. (This chapter presumes some familiarity with Prolog and the typed λ-calculi; respective introductions are Sterling & Shapiro [124] and Hindley & Seldin [63].)

3.1 The Logic Programming Framework

In general, logic programming languages offer several features relevant to formal program development and theorem proving:

- The underlying support for *unification* facilitates the implementation of rewrite and inference rules (*e.g.*, program transformations and theorem proving tactics). Typically, the application of such rules relies upon unification or *matching* — unification in which only one term's variables may be instantiated. Moreover, logic programming's support for unification is unobtrusive, allowing rules to be elegantly expressed.

- The logic programming *clause* was designed to encode inference rules: the head of the clause specifies the conclusion of a rule, while the body contains its premises.

Rewrite rules may also be elegantly expressed as clauses: the head typically consists of a predicate relating the rewrite's input and output, and the body specifies any necessary preconditions.

- The implicit *search* paradigm of depth-first backchaining is often sufficient for applying inference rules. And even for cases in which it is inadequate, the default search strategy can still aid specification, prototyping, and testing.
- The default search mechanism may be elegantly augmented with *programmer-defined control*, including guidance through user-interaction.

However, first-order logic programming languages, such as Prolog, suffer from other restrictions that make them less suitable for higher-order problem domains.

3.2 The Simply-typed λ-Calculus

λProlog replaces Prolog's first-order terms (*i.e.*, Herbrand terms[1]) with terms of the simply-typed λ-calculus. The λ-calculi are a family of languages introduced to study higher-order programming. Often the only data-type the pure forms of these languages contain is that of λ-terms — functions constructed with the binding operator λ. λ-calculi are nevertheless rich languages with which to experiment, in part because more complex data-structures may be encoded as functions [63].

Simple types. Before discussing λ-terms, we introduce the type system over λ-terms. Simple types may be inductively defined as

$$\tau \quad ::= \quad \mathbf{a} \mid A \mid \tau_1 \longrightarrow \tau_2 \mid \mathbf{a}\,\tau_1 \ldots \tau_n$$

where τ ranges over simple types, **a** over type constants, and A over type variables. *General notation:* We use boldface for constants (as well as for meta-variables such as **a**, which range over constants), and italics for variables (as well as for meta-variables such as τ). Function types are constructed with $\longrightarrow$, which associates to the right: $A \longrightarrow B \longrightarrow C$ is read as $A \longrightarrow (B \longrightarrow C)$. Type constructors consist of a type constant followed by some number of argument types (*e.g.*, **list int**).

Two predefined type constants of particular interest are **int** — the type of integers, and **o** — the type of λProlog propositions (goals and clauses). New type constants are defined by explicit declaration:

kind bool type.
kind list type → type.

---

[1] *Herbrand terms* (*i.e.*, those within the *Herbrand universe*) may be defined as

$$M \quad ::= \quad \mathbf{c} \mid \mathbf{f}\,M_1 \ldots M_n$$

where **c** ranges over first-order constants and **f** over function constants [124].

The above use of $\longrightarrow$ produces a *kind* rather than a type: types insure that λ-terms are well-formed, while kinds insure that the simple types themselves are well-formed [63].

λ-terms. Simply-typed λ-terms may then be inductively defined as

$$M \ ::= \ \mathbf{c} \mid x:\tau \mid MN \mid \lambda x:\tau.\, M$$

where M and N range over terms, $\mathbf{c}$ ranges over constants, x over variables, and τ ranges over simple types. The types of constant terms are separately declared within a type signature:

type succ int $\longrightarrow$ int.

A given λ-abstraction $\lambda x{:}\tau.M$ is of function type $\tau \longrightarrow \tau'$ provided M has type τ'. The juxtaposition MN denotes a λ-term application, which is of type τ' provided M is of type $\tau \longrightarrow \tau'$ and N is of type τ. λ-term application associates to the left: $\mathbf{abc}$ is read as $(\mathbf{ab})\mathbf{c}$. Thus the Prolog term $\mathbf{p(a,b)}$ is written as $\mathbf{p\ a\ b}$ in λProlog.

Type reconstruction. λ-terms become exceedingly redundant if all of the types required by the syntactic definition are explicitly included. A more succinct representation is afforded by eliding unnecessary type information. *Type reconstruction* is the process of rederiving those omitted types. In practice, all types are omitted from λProlog terms. Instead, the types of variables, untyped constants, abstractions, and applications are determined from context. In the sequel, we will omit types with the understanding that they are to be derived through type reconstruction.

Type reconstruction may fail for λ-terms that cannot be *well-typed* — *i.e.*, typed subject to the preceding rules. Similarly, type reconstruction may assign a lax or nonrestrictive type to λ-terms for which insufficient type information has been provided. Error and warning messages, respectively, are typically provided by λProlog implementations.

Polymorphism. The inclusion of type variables within simply-types offers a simple form of polymorphism. For example, the polymorphic identity function $\mathbf{id} \equiv \lambda x{:}A.\, x$ and double function $\mathbf{db} \equiv \lambda f{:}A \longrightarrow A.\, \lambda x{:}A.\, f(fx)$ are given the types

type id $A \longrightarrow A$.
type db $(A \longrightarrow A) \longrightarrow A \longrightarrow A$.

Particular occurrences of **id** or **db** may then be 'instantiated' (through the binding of A) to operate on distinct types, such as within **id** 1 and **id db**. The polymorphism of the simply-typed λ-calculus is also termed 'ML-style' or 'Milner-style' after the polymorphism of the language ML [90].

Basic operations on λ-terms. We use the notation $[N/x]M$ to denote the substitution of N for free occurrences of x in M. (Bound variables may have to be renamed to avoid capture; see below.) The term operations supported by λProlog include β- and η-reduction as well as α-conversion, which are defined as follows:

$(\lambda x.M)\,N$	$\Rightarrow_\beta\ [N/x]M$		*e.g.*, $(\lambda x.\lambda y.\mathbf{f}xy)y \Rightarrow_\beta \lambda y'.\mathbf{f}yy'$
$\lambda x.Mx$	$\Rightarrow_\eta\ M$	provided x not free in M	*e.g.*, $\lambda x.\mathbf{f}x \Rightarrow_\eta \mathbf{f}$
$\lambda x.M$	$\Rightarrow_\alpha\ \lambda y.[y/x]M$	provided y not free in M	*e.g.*, $\lambda x.x \Rightarrow_\alpha \lambda y.y$

Closures over these operations yield corresponding notions of λ-term convertibility: M is said to be β-convertible to M' if there exists a sequence of β-reductions and β-expansions (the inverse), applied at the top-level or to subterms, transforming M into M'. We may similarly define η- and α-convertibility, as well as combinations thereof. We use $=_{\alpha\beta\eta}$, or simply $=_{\beta\eta}$, to denote the equivalence relation of $\alpha\beta\eta$-convertible λ-terms.

In this calculus, $\beta\eta$-reductions are normalizing and Church-Rosser [63] — that is, maximal sequences of such reductions applied to a given well-typed λ-term terminate with a unique λ-term said to be in $\beta\eta$-normal form. This property is a consequence of the typing given to λ-terms, and is crucial for the unification algorithm, since the convertibility of two terms can be tested by comparing their normal forms for equivalence modulo the renaming of bound variables (α-conversion).[2]

Higher-order unification. *Unification* is the process of producing a common instance from two or more terms by instantiating either term's free variables with other terms.[3] We use the λProlog notation $M = N$ to indicate that the λ-terms M and N are to be unified. When unifying terms, we are typically interested in the *most general unifier* (MGU); for example, the MGU of $\mathbf{p}x$ and $\mathbf{p}y$ is simply $\langle x = y\rangle$, rather than the overly specific $\langle x = \mathbf{a}, y = \mathbf{a}\rangle$. (We shall continue to use $\langle\rangle$ to enclose *unification constraints.*)

Unification underlies the logic programming paradigm, but because λProlog terms are λ-terms, λProlog unification must be higher-order — *i.e.*, it must support the instantiation of variables to functions as well as to first-order constants. Terms of the λ-calculi, however, do not admit unique most general unifiers: consider that the unification of $f\mathbf{a} = \mathbf{caa}$ allows the variable f to be instantiated with any of $\lambda x.\mathbf{caa}$, $\lambda x.\mathbf{c}x\mathbf{a}$, $\lambda x.\mathbf{ca}x$, or $\lambda x.\mathbf{c}xx$, none of which is an instance of another (they are all closed). Thus, higher-order unification is inherently nondeterministic. Even worse, Goldfarb shows that higher-order (and in particular, second-order) unification is undecidable [49]. However, a semi-decision procedure effective in practice is presented by Huet [66] and extended by Elliott [39].

---

[2] Unlike $\beta\eta$-conversion, α-conversion is 'nondirectional.' Hence, in order to avoid expensive α-equivalence tests, it is necessary to employ representations that do not explicitly name bound variables — *e.g.*, de Bruijn indices [22].

[3] Knight presents an overview of unification research in [73]. For a formal treatment of first-order unification, see Lassez, Maher & Marriott [76].

3.3 Higher-order Language

A *domain* is said to be *higher-order* if it contains higher-order values — that is, values which take other values as arguments (*e.g.*, functions and predicates). For instance, the values manipulated by a higher-order programming language include functions. (By 'manipulated' we mean that functions are 'first-class' objects — *i.e.*, they can be bound to variables, passed as parameters, and returned from function calls.) Similarly, within a higher-order logic, the values that can be quantified include functions and predicates.

On the other hand, we consider a representation *language* to be *higher-order* if it contains a means for expressing argument binding: for example, the λ of λ-calculus or **lambda** in LISP. Such languages are particularly amenable to representing the values of higher-order domains, since the formulation of higher-order objects can be expressed with λ. We follow common practice in overloading the term 'higher-order' by applying it to values and domains (semantic entities), as well as languages (syntactic entities).

Higher-order domains. Many domains naturally involve binding constructs, and are thus best represented within higher-order languages: logics, programming languages, and natural language [106, 88, 85, 103]. This same need for higher-order representation also arises when one wants to reason 'at the meta-level' — that is, about λProlog programs. One would like facts (propositions) or properties (predicates) to be objects themselves. Prolog and other first-order representation languages allow this to some extent, but in a way that is only operationally, but not logically motivated. λProlog, on the other hand, facilitates higher-order programming — that is, the ability to create goals and programs, and pass them as arguments.

Binding operators. Within a higher-order language, binding operators are typically implemented via a single primitive such as λ. For example, the function $f(x) \equiv 2 * x$ might be represented simply as $f \equiv \lambda x.2 * x$. Similarly, $\forall x.\exists y.\, x < y$ might be expressed as **pi** $(\lambda x.$ **sigma** $(\lambda y.\, x < y))$. (In fact, this is the representation used within λProlog; the former is simply a more readable abbreviation.) The implementation of other binding operators in terms of λ allows $\alpha\beta\eta$-conversion and λ-term unification to be implemented once within the representation language rather than within individual client programs [106, 60]. Relegating such tasks to the representation language makes for more succinct, elegant programs.

Examples of λProlog. The higher-order predicate **select** of type

$$\textbf{type}\quad \textbf{select}\quad (A \longrightarrow \textbf{o}) \longrightarrow \textbf{list}\ A \longrightarrow \textbf{list}\ A \longrightarrow \textbf{o}.$$

may be encoded as

$$\begin{array}{llll} \textbf{select} & P & (x :: K) & (x :: L) \Leftarrow P\,x, \textbf{select}\ P\ K\ L. \\ \textbf{select} & P & (x :: K) & L \Leftarrow \textbf{select}\ P\ K\ L. \\ \textbf{select} & P & \textbf{nil} & \textbf{nil}. \end{array}$$

(As in Prolog ',' denotes conjunction. The symbol $\Leftarrow$ represents implication, and is equivalent to Prolog's :-. Finally, :: stands for the **cons** operation of LISP.) **select** $P\ K\ L$ insures that L is a sublist of K for which P holds. The following query, for example, selects the grandparents from the given input list:

$$?-\ \ \mathbf{select}\ \ (\lambda x.\exists y.\ \mathbf{parent}\ y\ x, \exists z.\ \mathbf{parent}\ z\ y)\ \ (\mathbf{tom} :: \mathbf{kate} :: \mathbf{van} :: \mathbf{leo} :: \mathbf{nil})\ \ L.$$

(To minimize the need for parentheses, $\lambda x.A\ x, \exists y.\ B\ x\ y$ is parsed as $\lambda x.(A\ x, \exists y.\ (B\ x\ y))$; that is, '.' binds very weakly to the right.)

Readers may argue that **select** could be formulated within Prolog simply by replacing $P\ x$ with **apply** $P\ x$, and further, that **grandparent** could itself be encoded as a top-level Prolog predicate:

$$\mathbf{grandparent}\ x \ \Leftarrow\ \mathbf{parent}\ y\ x, \mathbf{parent}\ z\ y.$$

In many situations, however, the 'inline' expression of higher-order arguments (such as the unnamed **grandparent**) is either necessary or desirable: for instance, reformulation as a clause is not applicable to higher-order functions, which are not predicates of type **o**. Moreover, first-order languages do not afford many operations over predicates, such as composition:

$$\mathbf{select\_or}\ \ P\ \ Q\ \ K\ \ L\ \Leftarrow \mathbf{select}\ (\lambda x.Px; Qx)\ K\ L.$$

(where the operator ';' represents inclusive 'or'.) Within Prolog, **select_or** cannot be programmed in terms of **select** (at least not without more complicated list operations); the closest approximation is the inequivalent

$$\mathbf{select\_or}\ \ P\ \ Q\ \ K\ \ L\ \Leftarrow \mathbf{select}\ P\ K\ L.$$
$$\mathbf{select\_or}\ \ P\ \ Q\ \ K\ \ L\ \Leftarrow \mathbf{select}\ Q\ K\ L.$$

First *vs.* higher-order. When higher-order values are represented with first-order terms, we often need 'new variables', need to check conditions such as 'where x does not occur in M', or must implement substitution in a way that 'renames bound variables if necessary.' In addition, procedures that depend upon the binding operator — *e.g.*, $\alpha\beta\eta$-conversion and higher-order unification — must be explicitly programmed. We claim that all this makes for a prohibitively complex encoding.

To justify such a conjecture, one might attempt to formalize a given higher-order example within a first-order language. However, at best such a strategy could only establish the inadequacy of one particular formulation. Arguing that first-order encodings are *generally* insufficient for higher-order domains is more problematic, because first-order languages certainly are expressively sufficient (being Turing equivalent). The proper question is, rather, one of expressive power and elegance — that is, are first-order languages sufficient to concisely and cleanly program over higher-order domains? While the issue remains open, as there exist no established criteria for making a determination, we remain convinced that higher-order expressivity is crucial for many higher-order domains.

Performance. The extent of the overhead incurred through higher-order operations remains unclear. On the one hand, implementation of $\alpha\beta\eta$-conversion and higher-order unification within the programming language is generally more efficient than user-programmed encodings, since an extra layer of language is avoided. On the other hand, the full power of undecidable higher-order unification is potentially too costly. Yet Huet's semi-decision procedure is effective in practice, simply because typical applications of λ-term unification are more restricted than the worst case.

A subset of λProlog named L_λ is currently being developed by Miller [84]. L_λ restricts higher-order unification to maintain the attractive properties of first-order unification: namely decidability and most general unifiers. The overhead of L_λ's restricted higher-order unification is not significantly different than that of first-order. We further discuss L_λ and its relevance to this thesis in §10.2.3.

3.4 λProlog

3.4.1 Clauses and Goals

Now we turn to the logical connectives of the language. λProlog expressions are distinguished based upon whether they appear as a goal G (*i.e.*, a query) or a program clause D (*i.e.*, a rule or fact), which are each of proposition type **o**. For Prolog, these two classes may be inductively defined as

$$\begin{array}{lcl} G & ::= & \mathbf{true} \mid A \mid G_1 \,,\, G_2 \mid G_1 \,;\, G_2 \\ D & ::= & \mathbf{true} \mid A \mid A \Leftarrow G \end{array}$$

where G ranges over goals (also termed G-formulas or G-forms), D over program clauses (or D-forms), and A over atoms. Atoms are propositional (in the Prolog case, Herbrand) terms of type **o** that do not have a logical operator at the top-level: that is, $\mathbf{c}\ (A\ ,\ B)$ is atomic, while $(\mathbf{c}\ A\ ,\ B)$ is not. By Prolog convention, variables within D are implicitly universal, while those within G are implicitly existential.

The above characterizes the Horn clauses [124], the logical basis of Prolog. Horn clauses may be generalized to higher-order Horn logic as follows:

$$\begin{array}{lcl} G & ::= & \mathbf{true} \mid A \mid G_1 \,,\, G_2 \mid G_1 \,;\, G_2 \mid \exists x\,[:\tau].\, G \\ D & ::= & \mathbf{true} \mid A \mid D_1 \,,\, D_2 \mid D \Leftarrow G \mid \forall x\,[:\tau].\, D \end{array}$$

where we have replaced Prolog's Herbrand terms with λProlog's simply-typed λ-terms. Atoms now take the form $\mathbf{p}\ M_1\ M_2..M_n$, where $\mathbf{p}$ is a predicate constant and $M_1\ M_2..M_n$ are its λ-term arguments (although $\beta\eta$-reduction may be required to reach this form).

λProlog is based upon the following generalization of higher-order Horn clauses:

$$\begin{array}{lcl} G & ::= & \mathbf{true} \mid A \mid G_1 \,,\, G_2 \mid G_1 \,;\, G_2 \mid D \Rightarrow G \mid \forall x\,[:\tau].\, G \mid \exists x\,[:\tau].\, G \\ D & ::= & \mathbf{true} \mid A \mid D_1 \,,\, D_2 \mid D \Leftarrow G \mid \forall x\,[:\tau].\, D \end{array}$$

Both $\Rightarrow$ and $\Leftarrow$ represent (intuitionistic) implication. Thus $G \Rightarrow D$ and $D \Leftarrow G$ stand for the same formula, where the latter is equivalent to Prolog's D :- G. Typically, a goal will be written as $D \Rightarrow G$, while a clause will be written as $D \Leftarrow G$.

The above classes define the core of λProlog— the higher-order Hereditary Harrop formulas [100], which generalize Horn clauses while preserving the basic character of a logic programming language. The logical operators have the following meanings and types (simple types are also employed to type logical expressions).

,	'and'	$\mathbf{o} \longrightarrow \mathbf{o} \longrightarrow \mathbf{o}$
;	'or' (inclusive)	$\mathbf{o} \longrightarrow \mathbf{o} \longrightarrow \mathbf{o}$
$\Rightarrow$	'implies'	$\mathbf{o} \longrightarrow \mathbf{o} \longrightarrow \mathbf{o}$
$\forall$	'for all'	$(A \longrightarrow \mathbf{o}) \longrightarrow \mathbf{o}$
$\exists$	'for some'	$(A \longrightarrow \mathbf{o}) \longrightarrow \mathbf{o}$

The types for $\forall x.\, P$ and $\exists x.\, Q$ arise from their respective representation as **pi** $\lambda x.\, P$ and **sigma** $\lambda y.\, Q$.

3.4.2 An Abstract Interpreter for λProlog

So that readers may arrive at a better understanding of λProlog, we herein provide an *informal* operational characterization of the language.[4] Then within §3.7, we offer a formal inference system.

Notation: First, we use $\mathcal{P}$ to denote an arbitrary logic program (set of D's). Next, $\mathcal{P} \vdash G$ denotes that G is a logical consequence of $\mathcal{P}$ — *i.e.*, that G follows (in the intuitionistic sense of logic programming) from $\mathcal{P}$. In order to speak about the state of the logic programming interpreter, we use $\mathcal{P} \Vdash G$ to represent the problem of solving G given the program $\mathcal{P}$. Thus, while $\mathcal{P} \vdash G$ expresses that G is logically valid given $\mathcal{P}$, $\mathcal{P} \Vdash G$ denotes that under a particular interpretation (*i.e.*, using a particular operational procedure for finding logic programming proofs) G is derivable from $\mathcal{P}$.

The table below contains a λProlog abstract interpreter, which consists of a series of backchaining search steps that follow the structure of the pending goal G.

true	$\mathcal{P} \Vdash \mathbf{true}$	
and	$\mathcal{P} \Vdash G_1 , G_2$	only if $\mathcal{P} \vdash G_1$ and $\mathcal{P} \Vdash G_2$.
or	$\mathcal{P} \Vdash G_1 ; G_2$	only if $\mathcal{P} \vdash G_1$ or $\mathcal{P} \Vdash G_2$.
augment	$\mathcal{P} \Vdash D \Rightarrow G$	only if $\{D\} \cup \mathcal{P} \Vdash G$.
instance	$\mathcal{P} \Vdash \exists x.\, G$	only if $\mathcal{P} \vdash [M/x]G$ for some λ-term M.
generic	$\mathcal{P} \Vdash \forall x.\, G$	only if $\mathcal{P} \vdash [\mathbf{c}/x]G$ for a new constant **c**.
backchain	$\mathcal{P} \Vdash A$	only if $(\forall \mathcal{X}.\, A_{\mathcal{X}} \Leftarrow G_{\mathcal{X}}) \in \mathcal{P}$, $A_{\mathcal{X}}$ unifies with A, and $\mathcal{P} \Vdash G_{\mathcal{X}}$.

[4] Our λProlog abstract interpreter is a slightly modified version of Nadathur & Miller's [100, p.813].

where $\mathcal{X}$ is a set of variables, $\forall\mathcal{X}$ represents quantification over that set, and $M_\mathcal{X}$ denotes a λ-term potentially containing free occurrences of $\mathcal{X}$.

For the sake of intelligibility, the abstract interpreter above ignores details essential for both correctness and efficiency:

- *Order of evaluation* — Conjunctive and disjunctive goals are 'evaluated' from left to right in a depth-first fashion. For the **or** case, the right branch need only be solved if the left fails (*i.e.*, is not satisfiable); for **and**, the right branch is only checked if solution of the left succeeds.

- *Type information* — To insure that λ-terms within G are properly typed, a type signature Σ must be generated for a given $\mathcal{P}$. Σ is extended as required by the **augment**, **instance** and **generic** operations.

- *Logical variables* — The **instance** strategy is realized by substituting a new logical variable of the appropriate type for the unknown λ-term. Subsequent computation may then fill in the unknown value (via unification).

- *Clause normal-form* — The reader may have noticed that the **backchain**, or rule application, step relies upon clauses of the form $\forall\mathcal{X}.\ A_\mathcal{X} \Leftarrow G_\mathcal{X}$, rather than the general form for D's given by the inductive definition. In §3.6 we illustrate a transformation nf that maps arbitrary D-forms into the normal-form D_{nf} — a conjunction of universally quantified rules of the form $A \Leftarrow G$. More formally, the structure of D_{nf} is defined as follows:

$$\begin{array}{rcl} D_{\mathsf{nf}} & ::= & D_\forall \mid D_{\mathsf{nf}}\ ,\ D_{\mathsf{nf}} \\ D_\forall & ::= & D_\Leftarrow \mid \forall x.\ D_\forall \\ D_\Leftarrow & ::= & A \Leftarrow G \end{array}$$

  The program $\mathcal{P}$ may in this way be mapped to a set of clauses of the form $\forall\mathcal{X}.\ A_\mathcal{X} \Leftarrow G_\mathcal{X}$. (An atomic clause A becomes $A \Leftarrow$ **true**.) Hence the test $\forall\mathcal{X}.\ A_\mathcal{X} \Leftarrow G_\mathcal{X} \in \mathcal{P}$ is correctly expressed as $\forall\mathcal{X}.\ A_\mathcal{X} \Leftarrow G_\mathcal{X} \in \mathsf{nf}(\mathcal{P})$.

  λProlog follows the Prolog convention of *universally closing* (*i.e.*, implicitly universally quantifying the free variables within) the D-forms of the original program. Clauses added to $\mathcal{P}$ via the **augment** step, however, remain open. Thus, $\mathcal{P} \vdash G$ potentially alters $\mathcal{P}$ by instantiating free variables of $\mathcal{P}$ in the course of solving G: for example, the λProlog query

$$?-\ \ \mathbf{p}\ x \Rightarrow (\mathbf{p}\ 1, \mathbf{p}\ 2).$$

  fails, as x is instantiated in the course of solving $\mathbf{p}\ 1$.

- *Clause unification* — The **backchain** step is implemented by unifying the pending atomic goal A with the head of a potentially applicable clause $\forall\mathcal{X}.\ A_\mathcal{X} \Leftarrow G_\mathcal{X}$. Like Prolog, this unification is accomplished by replacing the clause's universally quantified

variables $\mathcal{X}$ with new logical variables $\mathcal{Y}$ of the appropriate type. If $A\mathcal{Y} = A$, the interpreter attempts the solution of $\mathcal{P} \vdash G\mathcal{Y}$, where $G\mathcal{Y}$ is typically partially instantiated by the unification of A and $A\mathcal{Y}$.

- *Deterministic clause selection* — Finally, because the **backchain** strategy must have a means to enumerate clauses effectively, the logic program $\mathcal{P}$ is in reality a list rather than a set of D-forms. (The additions to $\mathcal{P}$ made through the **augment** step are effectively inserted at the head of this list.) A depth-first backtracking strategy is then taken to search for proofs of a given query.

- *Atoms with variable heads* — The higher-order Hereditary Harrop formulas disallow certain atoms (those occurring in *negative* positions) from being a variable predicate [100]. For example, the goal $xM_1..M_n \Rightarrow G$ is prohibited, because its solution could result in the assumption of a clause with a variable head (which would then be universally applicable). This restriction is not, however, strictly enforced by our abstract interpreter, nor for that matter within λProlog. Instead, goals and clauses may contain a variable predicate such as the assumption $xM_1..M_n$, so long as x is instantiated before the interpreter attempts to solve a goal $xM_1..M_n$ or assume a clause with head $xM_1..M_n$. This generalization is supported by current λProlog implementations.

3.4.3 λProlog Implementations.

This dissertation employs eLP, an implementation of λProlog developed by Conal Elliott and Frank Pfenning in the framework of the Ergo project at Carnegie Mellon University [38]. eLP is written in COMMON LISP and relies upon syntax tools of the Ergo Support System [77]. The examples presented within this thesis have each been run under eLP (or under an extended version developed herein).

A substantially more efficient implementation of λProlog within Standard ML is currently being completed by Pfenning. Yet another new λProlog implementation is being developed by Pascal Brisset at IRISA in France (`brisset@irisa.fr`).

3.5 Example: Lists & Mapping

In the next two sections, we present two more-extended examples of λProlog. The latter one, in particular, will be relevant to subsequent discussion.

The module within Figure 3.1 implements three simple operations over lists. List terms are built with the constructors **nil** and **::** (or 'cons'). Simple types for lists are produced by the **list** type constructor: for example, **list int** is the type of integer lists, while **list** A stands for homogeneous lists of an arbitrary type A. (Note that the latter does *not* admit inhomogeneous lists — *i.e.*, lists with elements of more than one type.)

The **member** function determines whether its first argument (of type A) occurs within its second argument (of type **list** A). The equivalence test employed by **member** is λ-term

```
module  lists.

kind    list      type ⟶ type.

type    nil       list A.
type    '::'      A ⟶ list A ⟶ list A.
type    member    A ⟶ list A ⟶ o.
type    append    list A ⟶ list A ⟶ list A ⟶ o.
type    nth       int ⟶ A ⟶ list A ⟶ o.

member  x  (x :: L)  ⇐  !.
member  x  (y :: L)  ⇐  member x L.

append  nil      K  K.
append  (x :: L) K  (x :: M)  ⇐  append L K M.

nth  0  x  (y :: L)  ⇐  !, x = y.
nth  n  x  (y :: L)  ⇐  m is n - 1, nth m x L.
```

Figure 3.1: List functions

unification. Cut (!) is a special side-effecting goal that commits the interpreter to all choices made since the selection of the clause containing the cut.[5,6] In **member**, ! prohibits looking further in the list once the given λ-term has been found. Herein then, ! is used primarily to improve efficiency rather than to change behavior.

The **append** predicate determines whether its first two list arguments may be appended to yield its third. Since **append** is defined as a predicate (in the 'logic programming style') rather than a function, it may be invoked with all arguments instantiated (as a 'check'), or with any pair instantiated, in which case **append** determines whether there exists a third list such that the 'append' relationship holds.

Finally, **nth** finds the nth element of a list.[7] The reason the first clause takes the given form rather than

```
nth  0  x  (x :: L)  ⇐  !.
```

is that the latter would continue down the list past the nth element y in the case that $x \neq y$.

The **lists** module could well have been presented within Prolog, as it does not make use of λProlog's higher-order features. The same cannot quite be said of the mapping functions contained within Figure 3.2. Of particular interest is the application of predicate P

[5]This includes λProlog's selection of alternative higher-order unifiers.

[6]For a thorough description of cut, see Sterling & Shapiro [124].

[7]The **is** construct represents assignment: For m **is** $n - 1$, m is set to the numeric value that results from evaluating $n - 1$. **is** differs from '=' in that the latter does not evaluate it arguments: $m = n - 1$ sets m to be the symbolic expression $n - 1$.

```
module  maps.
import  lists.

type    map_fun    (A ⟶ B) ⟶ (list A) ⟶ (list B) ⟶ o.
type    map_pred   (A ⟶ B ⟶ o) ⟶ (list A) ⟶ (list B) ⟶ o.
type    reduce     (A ⟶ B ⟶ B) ⟶ (list A) ⟶ B ⟶ B ⟶ o.
type    for_every  (A ⟶ o) ⟶ (list A) ⟶ o.
type    for_some   (A ⟶ o) ⟶ (list A) ⟶ o.

map_fun  f  nil      nil.
map_fun  f  (x :: L)  ((f x) :: K)  ⇐  map_fun f L K.

map_pred  P  nil      nil.
map_pred  P  (x :: L)  (y :: K)  ⇐  P x y, map_pred P L K.

reduce  f  nil      x  x.
reduce  f  (u :: L)  x  (f u y)  ⇐  reduce f L x y.

for_every  P  nil.
for_every  P  (x :: L)  ⇐  P x, for_every P L.

for_some  P  (x :: L)  ⇐  P x.
for_some  P  (x :: L)  ⇐  for_some P L.
```

Figure 3.2: Mapping functions

---

in **map_pred**, **for_every** and **for_some**, without Prolog's somewhat encumbering **apply** syntax. It is also important to note that the function f in **map_fun** and **reduce** is simply a λ-term; f is not a named procedure in the traditional sense of Prolog and other languages. (In higher-order languages such as λProlog, functions can be explicitly *constructed;* within first-order languages, functions can only be *simulated* as in **apply** $p\ x$.)

3.6 Example: Clause Normal-Form

The **nform** module of Figure 3.3 transforms arbitrary D-forms into the more restricted D_{nf} given in §3.4.2. Recall that D_{nf} is defined as

$$\begin{array}{rcl} D_{\mathsf{nf}} & ::= & D_{\forall} \mid D_{\mathsf{nf}}\ ,\ D_{\mathsf{nf}} \\ D_{\forall} & ::= & D_{\Leftarrow} \mid \forall x.\ D_{\forall} \\ D_{\Leftarrow} & ::= & A \Leftarrow G \end{array}$$

The nf mapping is required by the abstract interpreter presented in §3.4.2. In fact, nf is used by the eLP implementation for the clauses of the initial logic program $\mathcal{P}$, as well as the clauses added to $\mathcal{P}$ in the course of solving goals of the form $D \Rightarrow G$. Moreover, when we later extend λProlog with additional logical connectives, nf will serve as the basis of a

new clause normal-form. Finally, **nform** is illustrative of D-form and G-form manipulation in general, a recurring theme within more complex λProlog applications to come.

This normal-form conversion is captured by the following distributive transformations over D-forms:

$$
\begin{array}{lrcll}
(1) & \forall x.\,(D_1 x\ ,\ D_2 x) & \Longrightarrow & (\forall x.\,D_1 x)\ ,\ (\forall x.\,D_2 x) & \\
(2) & (D_1\ ,\ D_2) \Leftarrow G & \Longrightarrow & (D_1 \Leftarrow G)\ ,\ (D_2 \Leftarrow G) & \\
(3) & (\forall x.\,Dx) \Leftarrow G & \Longrightarrow & \forall x.\,(Dx \Leftarrow G) & \text{(provided } x \text{ not free in } G\text{).} \\
(4) & (D \Leftarrow G_1) \Leftarrow G_2 & \Longrightarrow & D \Leftarrow (G_1\ ,\ G_2) &
\end{array}
$$

Since D_{nf} requires conjunctions to be at the top-level, transformations (1) and (2) redistribute $\Leftarrow$ and $\forall$ inside of ','. Similarly, (3) redistributes $\Leftarrow$ inside of $\forall$. Finally, (4) collapses nested implications. Appropriate sequences of the above transformations map arbitrary D-forms to D_{nf}-forms. The proof is by induction over cases.

Before further exploring these transformations, there are two points to reinforce concerning λProlog's higher-order notation. First, when a bound variable x is not included as a potential argument to a variable function G as in $\forall x.\,(Dx \Leftarrow G)$, then x is *not* permitted to appear free within G; that is, the free variable restriction following (3) is already captured within the notation. Second, since $\forall x.\,Dx$ is represented as $\forall\,(\lambda x.Dx)$ where D is itself necessarily a λ expression, it makes since to express the whole simply as $\forall\,D$.

For example, through the above transformations the following D-forms on the left are mapped to the normal D-forms on the right:

$\mathbf{q}$	$\mathbf{q} \Leftarrow \mathbf{true}$
$\mathbf{p} \Rightarrow \mathbf{r} \Rightarrow \mathbf{q}$	$\mathbf{q} \Leftarrow (\mathbf{p},\ \mathbf{r})$
$\mathbf{p} \Rightarrow \forall x.\,(\mathbf{r}\,x \Rightarrow \mathbf{q})$	$\forall x.\,\mathbf{q} \Leftarrow (\mathbf{p},\ \mathbf{r}\,x)$
$\mathbf{p} \Rightarrow \forall x.\,(\mathbf{r}\,x \Rightarrow (\mathbf{q}, \mathbf{s}\,x))$	$\forall x.\,\mathbf{q} \Leftarrow (\mathbf{p},\ \mathbf{r}\,x)$, $\forall x.\,\mathbf{s}\,\mathbf{x} \Leftarrow (\mathbf{p},\ \mathbf{r}\,x)$

Within the code, **ndform** is the top-level routine; **requantify** redistributes a universal quantification 'below' top-level conjunctions; and **conjoin** redistributes $\Leftarrow$ below ',' and $\forall$, by adding the subgoal G to the preconditions of each nested D-form.

It is important to recognize that $\forall$ is used both as a data constructor to encode D-forms, and as a logical connective — for example, the $\forall x$ within

$$\textbf{conjoin}\ \ G\ \ (\forall D)\ \ (\forall D')\ \ \Leftarrow\ \ \forall x.\,\textbf{conjoin}\ G\,(Dx)\,(D'x)$$

If instead the above were simply

$$\textbf{conjoin}\ \ G\ \ (\forall D)\ \ (\forall D')\ \ \Leftarrow\ \ \textbf{conjoin}\ G\,(Dx)\,(D'x)$$

the variable x could be instantiated in the course of the computation. The explicit quantification acts as a guard ensuring that x remain universal.

```
module  nform.

type    ndform      o ⟶ o ⟶ o.
type    requantify  o ⟶ o ⟶ o.
type    conjoin     o ⟶ o ⟶ o ⟶ o.

requantify  (∀x. D1 x , D2 x)  (D1' , D2')  ⇐  !,  requantify (∀x. D1 x) D1',
                                                   requantify (∀x. D2 x) D2'
requantify  D                  D.

conjoin  G  (D1 , D2)      (D1' , D2')        ⇐  conjoin G D1 D1',
                                                 conjoin G D2 D2'.
conjoin  G  (∀D)           (∀D')              ⇐  ∀x. conjoin G (D x) (D' x).
conjoin  G  (A ⇐ true)     (A ⇐ G).
conjoin  G  (A ⇐ G1)       (A ⇐ (G , G1)).

ndform  (D1 , D2)  (D1' , D2')  ⇐  !,  ndform D1 D1',
                                       ndform D2 D2'.
ndform  (∀D)       D''          ⇐  !,  (∀x. ndform (D x) (D' x)),
                                       requantify (∀D') D''.
ndform  (D ⇐ G)    D''          ⇐  !,  ndform D D',
                                       conjoin G D' D''.
ndform  A          (A ⇐ true).
```

Figure 3.3: Clause normal-form conversion.

3.7 Inference System for λProlog

We may formalize the operational interpretation of λProlog given in §3.4.2 within an inference system. Our justification for doing this is twofold: (1) formal inference rules clarify the preceding informal description, and (2) such a system may be employed to prove properties of λProlog. The latter justification comes out of our need to establish the validity of λProlog extensions we propose in Chapter 5. The casual reader may skip this section, although the more formal discussions of Chapter 5 are then likely to be impenetrable.

Substitutions. A substitution is a mapping from variables to λ-terms. We represent the application of a substitution θ to an expression M (containing free variables) as θM, and we use $\psi\theta$ for the composition of substitutions $\psi \circ \theta$. Substitution binds less tightly than λ-term application; that is, $\theta MN = \theta(MN)$. Substitutions are applied to programs as well as to goals, because, as previously mentioned, $\mathcal{P}$ may contain free variables that become instantiated in the course of goal solution.

Recall the λ-term reduction rules defined in §3.2:

$$\begin{array}{lll} (\lambda x.M)\,N & \Rightarrow_\beta & [N/x]M \\ \lambda x.Mx & \Rightarrow_\eta & M \qquad \text{provided } x \text{ not free in } M \end{array}$$

As was stated then, for the simply-typed λ-calculus, maximal sequences of β and η reductions applied to a given λ-term terminate with a unique λ-term said to be in $\beta\eta$-normal form. For the remainder of this discussion, we assume that all λ-terms are $\beta\eta$-normal. This means that substitution θM and λ-term application MN are followed by $\beta\eta$-conversion to normal form.

Further notation: $\theta\backslash x$ means θ 'without' x (*i.e.*, $\theta\backslash x(x) = x$). Also, $\mathsf{free}(M)$ denotes the free variables of λ-term M, $\mathsf{dom}(\theta)$ represents the set of variables bound by substitution θ, and $\mathsf{ran}(\theta)$ is the set of λ-terms to which substitution θ maps $\mathsf{dom}(\theta)$. One substitution ψ is said to be an *instance* of another θ if for all M, there exists a further substitution σ such that $\psi M = \sigma\theta M$ — *i.e.*, θ is more general (or less defined) than ψ).

Finally, we say that a given substitution σ is *minimal*, or *most general*, with respect to a particular set of conditions, if θ satisfies those conditions, and if for any other substitution ψ also satisfying those conditions, ψ is a instance of θ.

The inference system. In order to more accurately speak about the state of a logic programming interpretation, we use $\mathcal{P} \vdash_\theta G$ to denote that G may be solved under $\mathcal{P}$ with a particular substitution θ. Figure 3.4 contains a list of inference rules formally defining the $\vdash$ relation. Following the structure of the abstract interpreter presented within §3.4.2, the inference system is 'goal-based' — the solution of a particular G-form (below the line) is reduced to the solution of one or more subgoals (above the line). For example, the rule

$$\frac{\mathcal{P} \vdash_\theta G_1 \qquad \theta\mathcal{P} \vdash_\psi \theta G_2}{\mathcal{P} \vdash_{\theta\psi} G_1 , G_2}$$

$$\frac{}{\mathcal{P} \vdash \mathbf{true}}$$

$$\frac{\mathcal{P} \vdash_\theta G_1 \qquad \theta\mathcal{P} \vdash_\psi \theta G_2}{\mathcal{P} \vdash_{\theta\psi} G_1 \, , \, G_2}$$

$$\frac{\mathcal{P} \vdash_\theta G_1}{\mathcal{P} \vdash_\theta G_1 \, ; \, G_2} \qquad \frac{\mathcal{P} \vdash_\theta G_2}{\mathcal{P} \vdash_\theta G_1 \, ; \, G_2}$$

$$\frac{\mathcal{P} \vdash_\theta Gy}{\mathcal{P} \vdash_\theta \forall x{:}\tau.\, Gx} \quad \text{where } y \notin \mathsf{free}(\theta G),\ y \notin \mathsf{free}(\theta\mathcal{P}),\ \text{and } y \notin \mathsf{dom}(\theta)$$

$$\frac{\mathcal{P} \vdash_\theta Gy}{\mathcal{P} \vdash_{\theta\backslash y} \exists x{:}\tau.\, Gx} \quad \text{where } y \notin \mathsf{free}(\theta G), \text{ and } y \notin \mathsf{free}(\theta\mathcal{P}).$$

$$\frac{\{D\} \cup \mathcal{P} \vdash_\theta G}{\mathcal{P} \vdash_\theta D \Rightarrow G}$$

$$\frac{(\forall \mathcal{X}.\, A\mathcal{X} \Leftarrow G\mathcal{X}) \in \mathsf{nf}(\mathcal{P}) \qquad \theta\sigma_{\mathcal{X}} A\mathcal{X} = \theta A \qquad \theta\mathcal{P} \vdash_\psi \theta\sigma_{\mathcal{X}} G\mathcal{X}}{\mathcal{P} \vdash_{\psi\theta} A}$$

where $\mathsf{dom}(\sigma_{\mathcal{X}}) \subseteq \mathcal{X}$,
$\mathsf{dom}(\theta) \cap \mathcal{X} = \emptyset$,
and $\sigma_{\mathcal{X}}$ & θ are most general.

Figure 3.4: Inference rules for λProlog.

describes the reduction of a conjunctive goal $(G_1 \, , \, G_2)$ into two subgoals, where G_1 is solved with substitution θ, and then θ is applied in the solution of G_2. Similarly,

$$\frac{\mathcal{P} \vdash_\theta Gy}{\mathcal{P} \vdash_\theta \forall x{:}\tau.\, Gx} \quad \text{where } y \notin \mathsf{free}(\theta G),\ y \notin \mathsf{free}(\theta\mathcal{P}),\ \text{and } y \notin \mathsf{dom}(\theta)$$

specifies that $\forall x{:}\tau.\, Gx$ is reduced to solving Gy for any y (since y is not bound by any substitution and does not appear free in $\mathcal{P}$).

While $\mathcal{P} \vdash_\theta G$ pertains to the particular inference system being defined, we use $\mathcal{P} \vdash G$ to instead denote that that there generally exists an *intuitionistic* proof of G given $\mathcal{P}$. Logic programming is *intuitionistic* or *constructive* in nature [87, 86]: for example, $\mathcal{P} \vdash G_1 \, ; \, G_2$ only if either $\mathcal{P} \vdash G_1$ or $\mathcal{P} \vdash G_2$. This disallows the derivation of *classical* tautologies such as $\vdash \mathbf{p} \, ; \neg\mathbf{p}$ and $\vdash (\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow (\neg\mathbf{p} \, ; \mathbf{q})$. (The $\neg$ operator stands for logical negation, which is *not* currently supported by λProlog.) The restriction to an intuitionistic system (as opposed

to a classical one) is so that the logic admits an effective proof procedure such as $\Vdash$. In fact, D-forms are restricted from containing disjunction (;) and existential quantification ($\exists$), precisely because of the difficulty in giving an operational interpretation to such clauses.[8]

3.7.1 Soundness

The inference system $\Vdash$ defined within Figure 3.4 is correct, or *sound*, in that only valid goals may be inferred: that is, $\mathcal{P} \Vdash_\theta G$ entails $\theta\mathcal{P} \vdash \theta G$. The soundness of $\Vdash$ may be proved by a straightforward induction (albeit tedious) over an arbitrary derivation $\mathcal{P} \Vdash_\theta G$. More precisely, given $\mathcal{P} \Vdash_\theta G$, we seek to show that $\theta\mathcal{P} \vdash \theta G$ by induction over the sequence of inference steps used to establish G:[9]

Basis.

Trivially, $\mathcal{P} \Vdash_\theta$ **true** implies $\theta\mathcal{P} \vdash$ **true**.

Induction step: if $\mathcal{P} \Vdash_\theta G$, then $\theta\mathcal{P} \vdash \theta G$. Proof is by cases.

Given $\mathcal{P} \Vdash_{\psi\theta} G_1 , G_2$.
By definition, $\mathcal{P} \Vdash_\theta G_1$ and $\theta\mathcal{P} \Vdash_\psi \theta G_2$.
From the *ind.hyp.* (*i.e.*, the induction hypothesis), $\theta\mathcal{P} \vdash \theta G_1$ and $\psi\theta\mathcal{P} \vdash \psi\theta G_2$.
Since $\psi\theta\mathcal{P} \vdash \psi\theta G_1$ follows from $\theta\mathcal{P} \vdash \theta G_1$, $\psi\theta\mathcal{P} \vdash (\psi\theta G_1 , \psi\theta G_2)$.
Hence $\psi\theta\mathcal{P} \vdash \psi\theta(G_1 , G_2)$.

Given $\mathcal{P} \Vdash_\theta G_1 ; G_2$.
By definition, $\mathcal{P} \Vdash_\theta G_1$ or $\mathcal{P} \Vdash_\theta G_2$.
From the *ind.hyp.*, $\theta\mathcal{P} \vdash \theta G_1$ or $\theta\mathcal{P} \vdash \theta G_2$.
Hence $\theta\mathcal{P} \vdash (\theta G_1 ; \theta G_2)$, and then $\theta\mathcal{P} \vdash \theta(G_1 ; G_2)$.

Given $\mathcal{P} \Vdash_\theta \forall x. Gx$.
By definition, $\mathcal{P} \Vdash_\theta Gy$ where $y \notin \mathsf{free}(\theta G)$, $y \notin \mathsf{free}(\theta\mathcal{P})$, and $y \notin \mathsf{dom}(\theta)$.
From the *ind.hyp.*, $\theta\mathcal{P} \vdash \theta Gy$.
Since $y \notin \mathsf{dom}(\theta)$, $\theta(Gy) = (\theta G)y$.
By *universal generalization* (applicable because of restrictions),

[8] For Horn logic and even higher-order Horn logic, intuitionistic and classical provability coincide [86], so the by-word 'intuitionistic' is only important for logics extended with embedded implication and embedded universal quantification, such as λProlog.

[9] A thorough soundness argument would also require that we establish the validity of λ-term substitution and $\beta\eta$-conversion. Instead, for the purposes of this discussion, we take as given that the underlying simply-typed λ-calculus operations preserve the soundness of $\vdash$.

$\theta\mathcal{P} \vdash \forall x.\,(\theta G)x$ for new variable x,
Hence $\theta\mathcal{P} \vdash \theta\forall x.\,Gx$.

Given $\mathcal{P} \vdash_{\theta\backslash y} \exists x.\,Gx$.
By definition, $\mathcal{P} \Vdash_\theta Gy$ where $y \notin \mathsf{free}(\theta G)$ and $y \notin \mathsf{free}(\theta\mathcal{P})$.
From the *ind.hyp.*, $\theta\mathcal{P} \vdash \theta Gy$.
From the restrictions on y, $(\theta\backslash y)\mathcal{P} \vdash ((\theta\backslash y)G)(\theta y)$.
By *existential generalization* (again applicable due to restrictions),
$(\theta\backslash y)\mathcal{P} \vdash \exists x.\,((\theta\backslash y)G)x$ for new variable x.
Hence $(\theta\backslash y)\mathcal{P} \vdash (\theta\backslash y)\exists x.\,Gx$.

Given $\mathcal{P} \vdash_\theta D \Rightarrow G$.
By definition, $\{D\} \cup \mathcal{P} \Vdash_\theta G$.
From the *ind.hyp.*, $\{\theta D\} \cup \theta\mathcal{P} \vdash \theta G$.
By *implication introduction*, $\theta\mathcal{P} \vdash \theta D \Rightarrow \theta G$.
Hence $\theta\mathcal{P} \vdash \theta(D \Rightarrow G)$.

Given $\mathcal{P} \vdash_{\psi\theta} A$.
By definition, there exists $D \in \mathsf{nf}(\mathcal{P})$ such that
$D = (\forall\mathcal{X}.\,A_\mathcal{X} \Leftarrow G_\mathcal{X})$, $\theta\sigma_\mathcal{X}A_\mathcal{X} = \theta A$, and $\theta\mathcal{P} \Vdash_\psi \theta\sigma_\mathcal{X}G_\mathcal{X}$,
$\mathsf{dom}(\sigma_\mathcal{X}) \subseteq \mathcal{X}$, and $\mathsf{dom}(\theta) \cap \mathcal{X} = \emptyset$.
From the *ind.hyp.*, $\psi\theta\mathcal{P} \vdash \psi\theta\sigma_\mathcal{X}G_\mathcal{X}$. (1)
Since $D \in \mathsf{nf}(\mathcal{P})$, it follows that $\mathcal{P} \vdash (\forall\mathcal{X}.\,A_\mathcal{X} \Leftarrow G_\mathcal{X})$,
which in turn has instance $\psi\theta\mathcal{P} \vdash \psi\theta(\forall\mathcal{X}.\,A_\mathcal{X} \Leftarrow G_\mathcal{X})$.
(This step relies upon the validity of our normal form mapping;
i.e., that for all $D' \in \mathsf{nf}(\mathcal{P})$, it is necessarily the case that $\mathcal{P} \vdash D'$.)
By *universal instantiation* (via $\sigma_\mathcal{X}$), it follows that $\psi\theta\mathcal{P} \vdash \psi\theta\sigma_\mathcal{X}(A_\mathcal{X} \Leftarrow G_\mathcal{X})$,
and then by *distributivity* of substitution, $\psi\theta\mathcal{P} \vdash \psi\theta\sigma_\mathcal{X}A_\mathcal{X} \Leftarrow \psi\theta\sigma_\mathcal{X}G_\mathcal{X}$. (2)
By *modus ponens* over (1) and (2), $\psi\theta\mathcal{P} \vdash \psi\theta\sigma_\mathcal{X}A_\mathcal{X}$.
From the definition, $\theta\sigma_\mathcal{X}A_\mathcal{X} = \theta A$.
Hence $\psi\theta\mathcal{P} \vdash \psi\theta A$.

3.7.2 Completeness

The dual theorem of soundness, *completeness*, typically fails for logic programs: even if $\theta\mathcal{P} \vdash \theta G$, the interpretation may fail to terminate in attempting to solve G. The source of this incompleteness is (1) the depth-first search of the logic programming paradigm, which may lead to infinite recursion in the solution of an otherwise valid goal (by selecting the 'wrong' branch of a disjunction or the 'wrong' clause from the program), and (2) λProlog's higher-order unification. One may, instead, speak of *nondeterministic* completeness — that is, given nondeterminism for logical and unification *choice points*, the remainder of the task

is complete. In fact, the inference rules of Figure 3.4 are nondeterministically complete as they make no commitment to order of evaluation. As this result is not particularly relevant to this thesis, we refer the interested reader to a similar discussion in Miller *et al.* [86].

Chapter 4

Explanation-Based Generalization

4.1 Generalization

Generalizations may be strictly syntactic, such as the replacing of a subexpression with a variable, or generalizations may be arbitrarily 'deep' semantically, such as making a rule applicable to a new set of situations. The space of possible generalizations is limited only by the expressiveness of the language in which results are phrased. For our purposes, the language of expressions and that of generalized expressions are one and the same — λProlog.

Inductive *vs.* analytical generalization. One class of approaches to the generalization problem is characterized as *inductive* or *similarity-based* [1, 30]. Similarity-based methods examine a set of instances of the desired concept. Typically, syntactic operations are then employed to derive a generalization covering those instances: *e.g.*, a pattern that matches each of them. Moreover, similarity-based approaches often make use of negative examples (*i.e.*, examples that are not instances of the desired concept) to keep the result from becoming *over-general.* We further discuss the inductive paradigm within §4.8.

The alternative approach to generalization is *analytical.* Analytical methods determine what and how to generalize by employing a theory of the problem domain. As a result, analytical methods generalize from a single example, rather than from the multiple instances required by inductive paradigms. To date, work on analytical techniques relies predominantly upon the mechanism of *explanation-based generalization* (EBG) and its variants [95, 25, 92, 40]. EBG abstracts a particular problem solution (*i.e.*, a proof or *explanation*), yielding an encapsulation of that solution — that is, a *derived rule* that more efficiently solves the original as well as related problems. While the proof-based generalizations of EBG are necessarily valid (with respect to the domain theory), similarity-based generalizations are guaranteed only to the extent that they cover the given examples.

Because the generalization space tends to be very large for any sufficiently rich language, similarity-based methods frequently employ a *bias* to determine how to generalize. (The simplest bias is to restrict the language in which results may be expressed.) One view of this bias is that it provides an analytical component to an otherwise similarity-based technique.

When this bias is so strong that a single example is sufficient for generalization, the method becomes analytical. Of particular interest currently is the combination of inductive and analytical methods [65, 114].[1]

Logic programming. Recently, logic programming has been used as a foundation for EBG [71, 108, 64, 8]. When explanation-based generalization is realized within the logic programming paradigm, the derivation of a goal produces as a byproduct the most general goal and the sufficient conditions for which the same sequence of proof steps (or logic programming search path) would apply. One argument put forward in favor of the logic programming framework is that it admits a common representation for all aspects of EBG: domain theory, training instance, query, derived rule, operationality criteria, *etc.* (These concepts are defined later in this chapter.) This helps in explicating the underlying principles in a uniform way and clarifies semantic issues.

One of the primary contributions of this thesis is the development of higher-order EBG, which is realized within the framework of λProlog. Our formulation of EBG is also unique in that it employs the modal logic operator □ to express the bias upon which EBG is founded.

4.2 First-order EBG

This section introduces first-order EBG within the logic programming framework. As it also introduces concepts unique to our formulation of EBG, it should be worthwhile even for readers familiar with the topic.

We begin by briefly illustrating explanation-based generalization with a first-order example from DeJong & Mooney [25, pp.158–166]: (We apologize to any readers offended by the morbidity of this example, but it has become standard in the literature.) EBG divides the theory of the problem domain between a *domain theory*, which we also denote with $\mathcal{D}$:

kill *a b* ⇐ **hate** *a b*, **possess** *a c*, **weapon** *c*.
hate *w w* ⇐ **depressed** *w*.
possess *u v* ⇐ **buy** *u v*.
weapon *z* ⇐ **gun** *z*.

and a *training theory* or $\mathcal{T}$:

depressed john.
buy john obj1.
gun obj1.

Both $\mathcal{D}$ and $\mathcal{T}$ are composed of λProlog clauses. For readers familiar with EBG, $\mathcal{T}$ roughly corresponds to training instance. Justification for the new terminology is given within §4.3.

[1] Hirsh, for instance, couples explanation-based and similarity-based methods by using version spaces (an inductive technique developed by Mitchell [93]) to generalize the results of EBG [65].

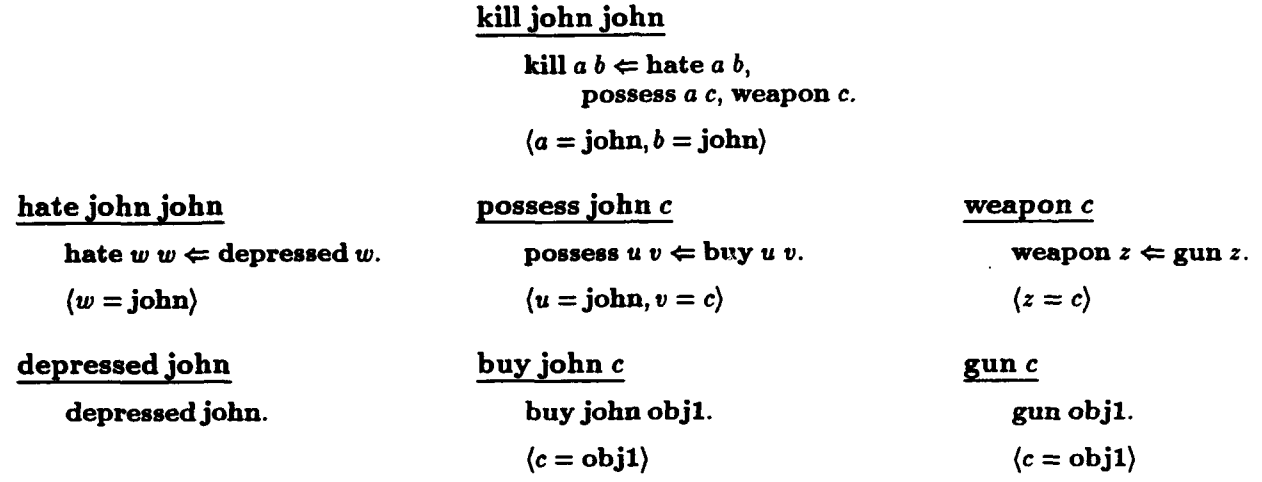

Figure 4.1: First-order proof.

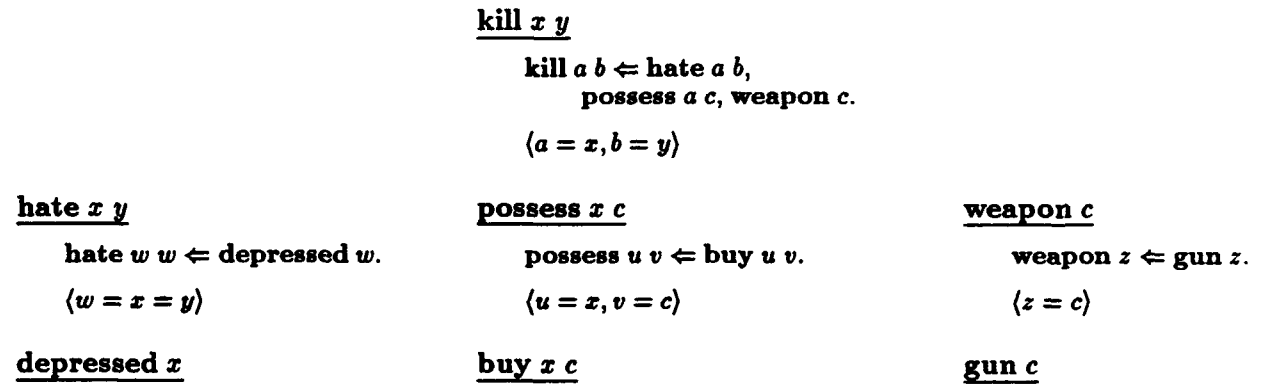

Figure 4.2: First-order generalized proof.

The EBG algorithm is additionally provided with a query, or goal, such as

? – **kill john john.**

EBG then requires a proof that solves the given query. Within the logic programming paradigm, such a proof may be expressed as a trace of λProlog computation. A proof of the above query is illustrated within Figure 4.1. Goals of the proof are underlined, while the program clause that reduces a particular goal appears underneath. In the course of applying each clause, its variables may be unified with constants or variables of the goal, resulting in the given unification constraints (enclosed in '⟨⟩').

EBG generalizes this explanation to produce an encapsulation of the employed proof strategy. In Figure 4.2, a generalized proof is constructed that corresponds to the original, except that clauses of $\mathcal{T}$ (or $\mathcal{T}$-clauses) are omitted. This forms EBG's bias in the generalization

space: the proof of the given query is generalized by abstracting steps involving clauses of the training theory. At the root of the new proof is a generalized query, which may be derived from the original by replacing each of the first-order constants with a variable: the goal **kill john john** becomes the general goal **kill** x y. Clauses of $\mathcal{D}$ (or $\mathcal{D}$-clauses) applied in the first proof are correspondingly applied in the second. This restricts the outcome by propagating unification constraints through the proof (*e.g.*, **kill** x y becoming **kill** x x). Leaves of the generalized proof (*e.g.*, **gun** c) correspond to subgoals of the original proof that were derived from $\mathcal{T}$. These leaves are accumulated in a conjunction of conditions sufficient to establish the generalized query:

kill x x $\Leftarrow$ **depressed** x, **buy** x c, **gun** c.

We will frequently refer to the resulting proof encapsulation, as a *derived rule*, or as an *explanation-based generalization*, or simply as a *generalization.*

4.3 Modal Logic

Our formulation of EBG depends upon the separation of $\mathcal{D}$ and $\mathcal{T}$, since only rules of the former are incorporated within generalized proofs. To differentiate the two, we prefix $\mathcal{D}$-clauses with the $\Box$ operator, which is borrowed from *modal* logic — logics in which propositions have multiple levels or modes of truth, such as 'may be' and 'must be.' [2]

We illustrate our use of $\Box$ on the first-order example of §4.2. $\mathcal{D}$ and $\mathcal{T}$, which constitute the logic program, may now be jointly expressed as

$\Box\ \forall a\ \forall b\ \forall c.$ **kill** a b $\Leftarrow$ **hate** a b, **possess** a c, **weapon** c.
$\Box\ \forall w.$ **hate** w w $\Leftarrow$ **depressed** w.
$\Box\ \forall u\ \forall v.$ **possess** u v $\Leftarrow$ **buy** u v.
$\Box\ \forall z.$ **weapon** z $\Leftarrow$ **gun** z.
depressed john.
buy john obj1.
gun obj1.

The above presentation does *not* rely upon λProlog's implicit universal quantification of a program's logical variables. This is because our EBG algorithm differentiates between the clauses $\Box\ \forall x.\ D$ and $\forall x.\ \Box\ D$. (The motivation for this distinction may be found in §8.9.) However, since explicitly specifying quantifiers can become exceedingly tedious, we introduce the '!!' shorthand to represent this universal quantification implicitly. The first clause of $\mathcal{D}$ may then be expressed as

!! **kill** a b $\Leftarrow$ **hate** a b, **possess** a c, **weapon** c.

And, for the query **kill john john**, the resulting explanation-based generalization becomes

!! **kill** x x $\Leftarrow$ **depressed** x, **buy** x c, **gun** c.

[2] For an introduction to modal logic, see Chellas [16].

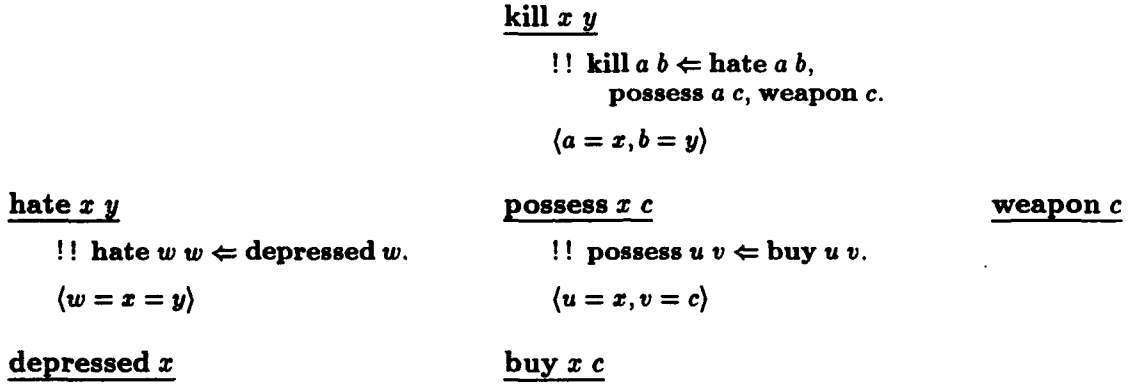

Figure 4.3: Less specific generalized proof.

Traditionally, the modal operator □ (sometimes called 'L') precedes *necessarily* true sentences, or equivalently, those true in 'all possible states' or at 'all times.' Non-prefixed sentences are only *contingently* true, true in the 'current state' or at the 'current time.' Our incorporation of □ is founded upon a correspondence between (1) EBG's separation of domain and training theory and (2) modal logic's separation of necessary and contingent truth: Because the validity of the generalizations derived through EBG depend solely upon $\mathcal{D}$, more stringent truth requirements are placed upon $\mathcal{D}$-clauses — namely that they be true in all possible configurations of the problem space being modeled. Clauses of $\mathcal{T}$, as they are excluded from generalized proofs, can safely be revised or removed without invalidating the derived generalizations (*e.g.*, **depressed john** becoming false). Such revision could be explained semantically as 'changing states' or 'switching worlds.'[3]

Suppose that within the suicide example, we remove the □ from the clause **weapon** $z \Leftarrow$ **gun** z. This results in the generalized proof of Figure 4.3, and the generalization

!! **kill** x $x \Leftarrow$ **depressed** x, **buy** x c, **weapon** c.

The above rule is more general than the previous one, but its application requires more computation. This illustrates the trade-off inherent in the partitioning of $\mathcal{D}$ and $\mathcal{T}$: $\mathcal{D}$-clauses get 'compiled into' the rules derived through EBG, while $\mathcal{T}$-clauses must be evaluated at 'runtime' (the time of application).

Now suppose instead that we replace the last clause with □ **gun obj1**, again within the original example. This has the effect of 'anchoring' the generalization to **obj1**, with the result of an identical query being the generalized proof of Figure 4.4 (whose rightmost branch is solved), and the generalization

!! **kill** x $x \Leftarrow$ **depressed** x, **buy** x **obj1**.

---

[3]Readers familiar with EBG may wonder how the concept of *operationality* relates to $\mathcal{D}$ and $\mathcal{T}$. We defer this discussion until §4.5.

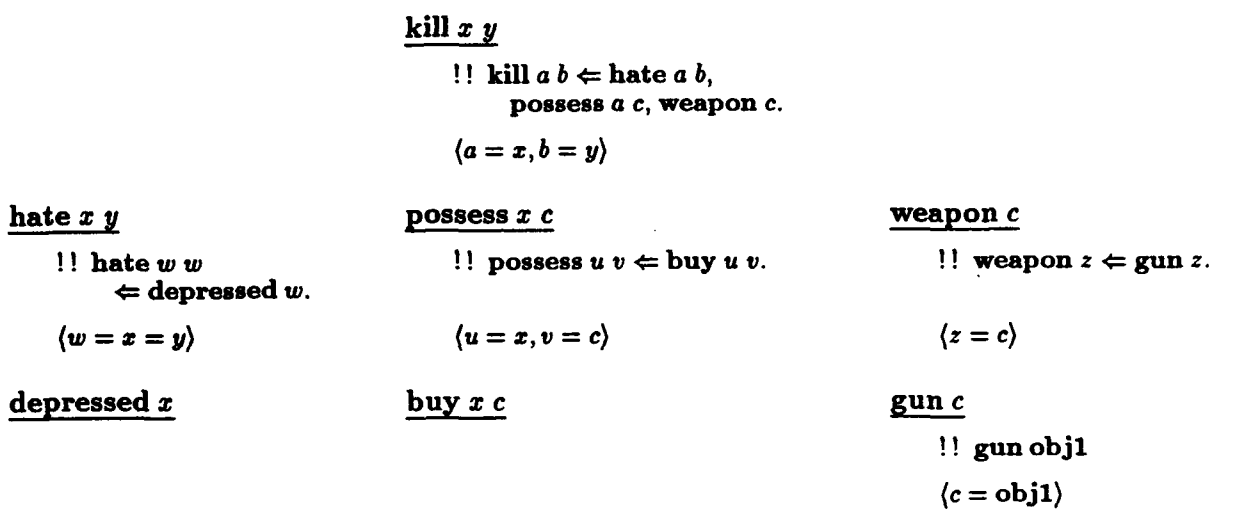

Figure 4.4: More specific generalized proof.

By moving a clause from $\mathcal{T}$ to $\mathcal{D}$, we make the resulting generalization more specific. Such a shift is, however, dangerous in that the generalization then depends upon the validity of **gun obj1**. In another configuration where **obj1** is not a gun, the derived rule is false!

$\Box$ may also be used in the query language to determine the 'necessary' truth of a goal G, but this is less likely to yield interesting generalizations as the proof of $\Box\ G$ is composed solely of $\mathcal{D}$-clauses (and is therefore isomorphic to the generalized proof). Nevertheless, the derived rule may be a generalization in that constants of G are abstracted.

Training instance. Previous realizations of EBG have used the term 'training instance' rather than our 'training theory' $\mathcal{T}$. While the literature makes the same operational distinction of excluding training instance from generalized proofs, the term additionally carries the connotation of embodying a single example situation from which the learner should generalize. We have taken the liberty of renaming the training instance to avoid that connotation.

Typically within logic programming implementations of EBG, atomic clauses are directly recognized as belonging to the training instance [71, 64, 108] — *e.g.*, **gun obj1**. Although this notion of training instance offers some intuitive value, we find it artificially restrictive. There exist atomic clauses that we might want to include within $\mathcal{D}$: !! **adjacent** x x.[4] The same is true even for constant atomic clauses: for example, to represent that **block1** is glued to the table we could assert $\Box$ **on block1 table**. Alternatively, we might want to include variables and logical connectives within $\mathcal{T}$-clauses: for example, under the temporary condition that all blocks are stacked in two-high pairs, we might assert $\forall x \forall y$. **on** x $y \Rightarrow$ (**clear** x, **on** y **table**). $\Box$ furthermore affords the potential to intermix knowledge of the domain and training theory through the nesting of $\Box$ below the top-level of clauses.

---

[4]To accomplish this, some EBG systems employ the trick of writing the clause as **adjacent** x $x \Leftarrow$ **true**.

Our use of $\Box$, then, avoids what we believe to be undue limitations on the training instance: our training theory may instead contain arbitrary λProlog clauses. $\Box$ provides an underlying limit to the generalization by allowing overly specific knowledge to be excluded from the derived rules. Within a nonmonotonic logic, for example, such a mechanism could guarantee validity for the resulting generalizations by distinguishing the fixed theory from temporal assertions. In addition to providing greater expressiveness, a modal logic representation for the distinction between $\mathcal{D}$ and $\mathcal{T}$ can be given a clear semantics that is independent of a particular search procedure or generalization algorithm.

Modal logic and EBG. Admittedly, the analogy that contingency is to necessity as training theory is to domain theory is philosophically questionable. The basis for our incorporation of $\Box$ is rather that the operator elegantly *models* the difference between $\mathcal{T}$ and $\mathcal{D}$ in a formal (as opposed to an operational) manner — that is, through a formal language and the accompanying proof system. Our use of the terms 'contingency' and 'necessity' is meant to convey some semantic intuition about why $\Box$ models this distinction. One could easily turn this observation around and say that we have found yet another interpretation of $\Box$.[5]

The inclusion of $\Box$ within λProlog leads to a rich language for higher-order EBG — $\lambda^{\Box}$Prolog. The remainder of this chapter continues the discussion of modal logic and higher-order EBG; within Chapters 5 & 8 we further develop and formalize $\lambda^{\Box}$Prolog.

4.4 Higher-order EBG

In Chapter 3 we made the case for the additional expressiveness afforded by higher-order language, and in particular for λProlog. Expressive elegance is intimately tied to effective generalization: If knowledge is represented in an inappropriate language, then it is less likely that the desired generalizations can be expressed in a natural and concise manner, and also less likely that they can even be found. In particular, the cumbersome encoding of higher-order domains within first-order languages inhibits reasoning and generalization. But to date, the application of EBG has been limited to first-order languages. To facilitate EBG's application to higher-order domains, we extend the technique to *higher-order explanation-based generalization* — that is, EBG in which functions and predicates as well as first-order constants may be abstracted, or replaced with variables.

We would like to assert more — namely that first-order encodings are inadequate for the task of generalization over higher-order domains, in particular because primitive syntactic manipulations inevitably intrude into the generalizations. This is, however, simply one aspect of the open argument between first- and higher-order languages (§3.3).

---

[5]There are already many such interpretations: $\Box$ can stand for 'formally provable', or for truth in all reachable worlds in a Kripke semantics [69], *etc.*

We introduce higher-order EBG with another example frequently exploited within the literature, that of symbolic integration. Calculus integration is fundamentally a higher-order domain in that the items being manipulated are functions, and functions and variables that range over functions are not naturally part of a first-order language. Consider the following higher-order rules for integration: the first treats exponentiation, the second extracts a constant factor, and the third splits a sum. The predicate **intgr** relates a function to its indefinite integral. To increase readability, we use a mathematical notation for arithmetic operators not included in λProlog — in particular, exponentiation and division.

$$
\begin{array}{llll}
!!\ \textbf{intgr} & (\lambda x.x^a) & (\lambda x.x^{a+1}/(a+1)). & \\
!!\ \textbf{intgr} & (\lambda x.a*fx) & (\lambda x.a*hx) & \Leftarrow\ \textbf{intgr}\ f\ h. \\
!!\ \textbf{intgr} & (\lambda x.fx+hx) & (\lambda x.f'x+h'x) & \Leftarrow\ \textbf{intgr}\ f\ f', \textbf{intgr}\ h\ h'. \\
\textbf{intgr} & \cos & \sin .
\end{array}
$$

The traditional binding notation of dx has been replaced with λ-terms. Missing from the first rule is the restriction that $a \neq -1$, because λProlog does not admit constraints other than those imposed by unification.[6] Readers may find the last rule more intelligible in its η-expanded form **intgr** $(\lambda x.\cos\ x)\ (\lambda x.\sin\ x)$. The cosine rule is an example a $\mathcal{T}$-clause that is not 'contingent': while the rule is just as valid as the others, it represents a proof step we wish to abstract under EBG.

The query

$$?-\ \ \textbf{intgr}\ \ (\lambda x.3*x^2+\cos x)\ \ h.$$

yields the solution

$$h = \lambda x.3*x^{2+1}/(2+1)+\sin x$$

and the generalization

$$!!\ \textbf{intgr}\ \ (\lambda x.a*x^b+fx)\ \ (\lambda x.a*x^{b+1}/(b+1)+f'x)\ \ \Leftarrow\ \ \textbf{intgr}\ \ f\ \ f'.$$

The proof and generalized proof associated with this example are given in Figures 4.5 and 4.6, respectively.

The generalization space of higher-order EBG is significantly larger than that of first-order, in that higher-order constants are additionally subject to variable replacement: consider that in the first-order case of Figure 4.2, the goal **kill** x y is fully general, while for higher-order, a single variable G ranging over goals is fully general. Also unlike the proofs of §4.2 & 4.3, the integration proofs make use of higher-order unification, which implicitly enforces the restrictions placed upon free and bound variables: for example, within an application of the power rule, $\lambda x.x^a$ will not unify with $\lambda x.x^x$ since a may not contain free occurrences of x. What is more, function variables may in this way appear in the derived generalizations (*e.g.*, f).

[6] For a discussion of logic programming with constraints, see Jaffar & Lassez [70].

intgr $(\lambda x.3 * x^2 + \cos x)$ r

!! **intgr** $(\lambda x.gx + fx)$ $(\lambda x.g'x + f'x)$
$\Leftarrow$ **intgr** g g', **intgr** f f'.

$\langle g = \lambda x.3 * x^2, f = \cos, r = \lambda x.g'x + f'x \rangle$

intgr $(\lambda x.3 * x^2)$ g'

!! **intgr** $(\lambda x.a * hx)$ $(\lambda x.a * h'x)$ $\Leftarrow$ **intgr** h h'.

$\langle a = 3, h = \lambda x.x^2, g' = \lambda x.a * h'x \rangle$

intgr cos f'

intgr cos sin.

$\langle f' = \sin \rangle$

intgr $(\lambda x.x^2)$ h'

!! **intgr** $(\lambda x.x^b)$ $(\lambda x.x^{b+1}/(b+1))$.

$\langle b = 2, h' = \lambda x.x^{b+1}/(b+1) \rangle$

Figure 4.5: Higher-order proof.

G

!! **intgr** $(\lambda x.gx + fx)$ $(\lambda x.g'x + f'x)$
$\Leftarrow$ **intgr** g g', **intgr** f f'.

$\langle G = \textbf{intgr}\ (\lambda x.gx + fx)\ (\lambda x.g'x + f'x) \rangle$

intgr g g'

!! **intgr** $(\lambda x.a * hx)$ $(\lambda x.a * h'x)$ $\Leftarrow$ **intgr** h h'.

$\langle g = \lambda x.a * hx, g' = \lambda x.a * h'x \rangle$

intgr f f'

intgr h h'

!! **intgr** $(\lambda x.x^b)$ $(\lambda x.x^{b+1}/(b+1))$.

$\langle h = \lambda x.x^b, h' = \lambda x.x^{b+1}/(b+1) \rangle$

Figure 4.6: Higher-order generalized proof.

Related work. Donat & Wallen also address the step from first- to higher-order EBG over the domain of symbolic integration [37]. Higher-order generalization allows extremely general rules to be extracted from particular problem solutions. Our work focuses on how to control EBG to avoid over-general generalizations, and yet at the same time take advantage of the higher-order nature of the language. Donat & Wallen's work concentrates on how one could still usefully apply very general learned rules. To that end they introduce some control constructs into the higher-order unification process.[7] In that sense our approaches differ fundamentally.

Donat and Wallen's approach also utilizes a first-order representation of integrals (from which they then produce higher-order generalizations). This first-order encoding requires additional constraints, manifest in the **constant** primitive, which pervade their derived rules and which are avoided under our higher-order encoding.

4.5 Operationality

We illustrated in §4.3 how □ defines which proof steps are included in generalized proofs. Within the EBG paradigm, the traditional means of restricting the extent of generalized proofs is through *operationality criteria*: By establishing that a particular goal meets an operationality criterion, the subtree deriving it is 'pruned' from the generalized proof. That is, an operationality criterion can be viewed as a predicate that determines whether a given goal should be a leaf of the generalized proof. The term 'operational' arises from the requirement that such subgoals be easily derivable, since operational subgoals must be established (solved) in the course of applying an explanation-based generalization.

To illustrate, if we augment the original formulation of the suicide example (§4.3) with a declaration that the goal **weapon** z is operational, the EBG algorithm produces the derived rule

$$!!\ \mathbf{kill}\ x\ x \Leftarrow \mathbf{depressed}\ x, \mathbf{buy}\ x\ c, \mathbf{weapon}\ c.$$

This follows because the computation below the operational goal **weapon** z is herein excluded from the generalized proof. Thus, while □ establishes which branches of the proof tree will lead to antecedents in the derived rule (because they are established by contingent clauses), operationality criteria determine to what depth to which the proof tree extends within a branch.[8]

Although □ and operationality criteria are both mechanisms that limit the extent of generalized proofs, the former is a property of clauses (*i.e.*, whether or not they contain □), while the latter is a property of goals (*i.e.*, whether or not they are operational).

---

[7] In particular, they permit *filter expressions* to be defined to preclude trivial higher-order unifications; for example, in matching $f(\mathbf{a}) = \mathbf{a} + \mathbf{b}$, the instantiation of $f = \lambda x.\mathbf{a} + \mathbf{b}$ might be less desirable than $f = \lambda x.x + \mathbf{b}$.

[8] Keller reviews existing formulations of operationality, and develops the topic significantly beyond its treatment herein [72].

Operationality criteria present the same trade-off we have seen for □: the closer the operational subgoals are to the root of the generalized proof, the more generally applicable the derived rule is, but also the more work is required to apply it. In fact, one reason that the partition between domain and training theory has not received more consideration within the literature is that the mechanism of operationality typically precludes any $\mathcal{T}$-clauses from entering into generalized proofs.

Operationality criteria, however, do provide features beyond the capabilities of □. For instance, they offer a generally more concise means to define generalized proofs: by declaring only a single subgoal to be operational, the entire branch of the generalized proof underneath is excluded, or pruned, from the generalization. Achieving the same effect with □ alone would require removing from $\mathcal{D}$ each of the program clauses applied within that branch. Moreover, if a particular rule is used pervasively in a proof, it might be necessary to include it within both $\mathcal{D}$ and $\mathcal{T}$ (and then use some form of additional control to discriminate between occurrences.) Operationality criteria do not present a corresponding problem, as it is unlikely that recurring subgoals should be considered both operational and non-operational.

□ does, on the other hand, offer a means by which to generalize in a entirely different manner: consider that even interior steps can be abstracted from generalized proofs via □. We illustrate this with one last contrived perturbation of the suicide example:

!! **kill** *a b* ⇐ **hate** *a b*, **possess** *a c*, **weapon** *c*.
!! **hate** *w w* ⇐ **depressed** *w*.
!! **possess** *u v* ⇐ **buy** *u v*.
weapon *z* ⇐ **gun** *z*.
!! **gun** *z* ⇐ **pistol** *z*.
depressed john.
buy john obj1.
pistol obj1.

For the standard query **kill john john**, the above theory leads to the generalized proof in Figure 4.7, and the generalization

!! **kill** *x x* ⇐ **depressed** *x*, **buy** *x c*, **pistol** *c*, (**gun** *c* ⇒ **weapon** *c*).

Note that this rule allows the use of an arbitrarily extended computation to establish the subgoal **gun** *c* ⇒ **weapon** *c*.

We conclude that the mechanisms of operationality criteria and □ are complementary, and while □ is sufficient to formulate the examples presented within this dissertation, we do not suggest it as a replacement for operationality criteria. In fact, the combination of the two is particularly attractive: modal logic induces an underlying limit to the specialization of derived rules that potentially prohibits EBG from yielding 'incorrect' generalizations. Operationality criteria, in turn, provide a means to 'fine tune' selection from the space of possible generalizations admitted by □. This is particularly true of *dynamic* operationality criteria — *i.e.*, those which allow the operationality of goals to be defined and redefined within the computational framework [64]. Since dynamic operationality criteria are subject

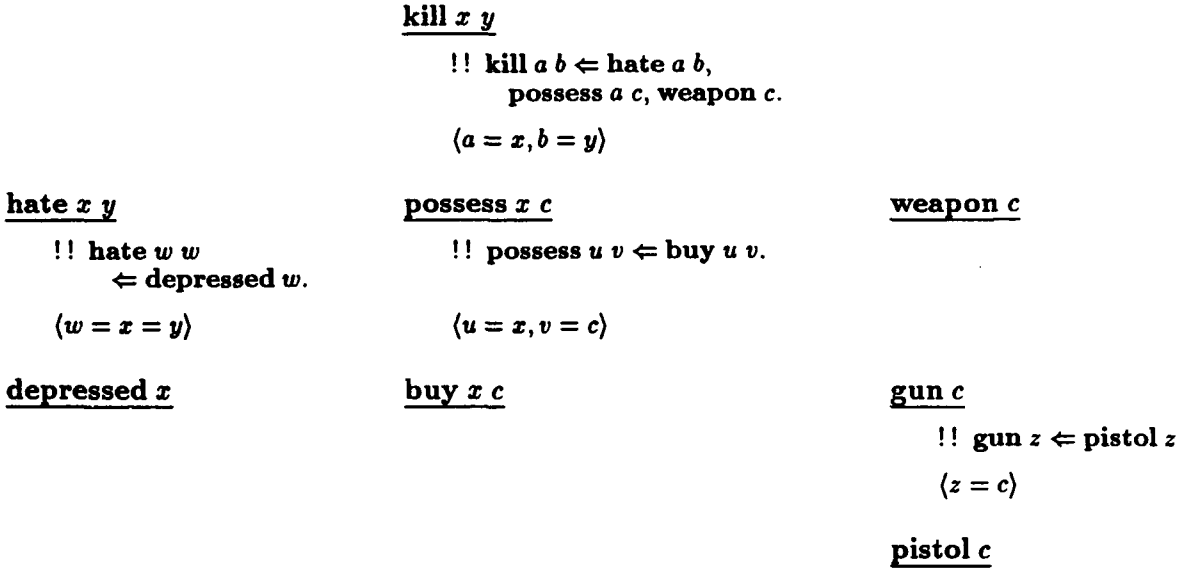

Figure 4.7: Generalized proof with internal step abstracted.

to change, derived rules typically incorporate some representation of the utilized criteria to insure their continued integrity. Through the use of □, the need for these *operationality preconditions* is reduced or eliminated because of the guarantee of validity afforded by □. We further compare and contrast operationality and □ in §6.4.

4.6 Partial Evaluation *vs.* EBG

As pointed out by Van Harmelen & Bundy [129], explanation-based generalization is closely related to partial evaluation (PE). On the surface the two seem to be very different paradigms: consider that EBG is a process of generalization, while PE is one of *specialization.* These opposing definitions may be reconciled by considering two different views of EBG: The first, and the one thus far articulated, is that EBG involves the generalization of a particular solved problem (by abstracting those solution steps based upon $\mathcal{T}$-clauses or those solving operational goals). However, one may alternatively view EBG as as a specialization (or *partial evaluation*) of $\mathcal{D}$ to the logic programming computation solving the given goal. In fact, each of the explanation-based generalizations we produce could equally be derived through partial evaluation.

In spite of this correspondence, the mechanisms by which PE and EBG produce results differ in fundamentally important ways:

- EBG relies upon an example *explanation* — *i.e.*, a goal and the accompanying solution, while PE works from an unsolved general goal.

- EBG employs a bias in the logic program ($\mathcal{D}$ *vs.* $\mathcal{T}$ and/or operationality criteria) to determine the form of generalized proof, while PE utilizes some form of search control to explore possible expansions of a general goal.

In short, EBG is a specialized application of partial evaluation in which search control is provided by a particular example, the partition between $\mathcal{D}$ and $\mathcal{T}$, and the operationality criteria.

4.6.1 Other Work

Etzioni's thesis [41] considers the replacement, within the framework of Prodigy [92], of EBL with Static, which instead uses a formal analysis of the domain theory incorporating partial evaluation to derive new rules. For the domains he considers, Static generally outperforms EBL; that is, through its additional analysis, his system is able to derive better rules. We are skeptical, however, as to whether this approach can be effectively extended to intractable domains wherein the domain theory itself is relatively small (*e.g.*, a set of axioms), but for which the space of possible partial evaluations (*i.e.*, the closure of that domain theory) is infinite. For such theories, some form of search control, such as is provided by EBG's example, appears indispensable.

4.6.2 Example: "peval"

In this section we present a rudimentary partial evaluator for λProlog (not $\lambda^{\Box}$Prolog) because it raises important issues in the difference between EBG and PE, and also because it will be pertinent to later discussion. An unabridged partial evaluator, as well as an extended example of partial evaluation, is included within Appendix A.3.

Figure 4.8 contains the code for **peval**, where its second argument is the result of partial evaluating its first. The consequence of **peval** E G is that the derived rule $E \Leftarrow G$ follows from the program. (The new class E stands for λProlog logical expressions that are both clauses and goals — *i.e.*, $E = D \cap G$. If the potential result of PE, $E \Leftarrow G$, is actually to be added to the program, then E must additionally be a legal D-form.)

Search control is provided by the user, who determines (1) whether PE is to continue at each atomic goal G_a (via **stop?**), (2) whether a particular applicable rule from the program-base is to be applied in the solution of G_a (via **apply_rule?**), and (3) which branch of an alternation to partially evaluate (via **left?**).[9] The predicate **hyp** is used to enumerate clauses of the program-base.[10] These clauses are assumed to be in normal-form D_{nf}, which we defined in §3.6.

---

[9]Within λProlog's input predicate **read** $\lambda x.Gx$, the variable x is bound to the entered term before execution of **read**'s body Gx. **read** $\lambda q.q$, then, provides a simple method for querying the user for a yes/no (*i.e.*, true/false) response.

[10]The need for **hyp** is further discussed in §8.4.1.

```
peval  true        true          ⇐ !.
peval  (G1 , G2)   (G3 , G4)     ⇐ !, peval G1 G3, peval G2 G4.
peval  (G1 ; G2)   G             ⇐ !, ((left? (G1 ; G2), !, peval G1 G); peval G2 G).
peval  (∀x. Gx)    (∀x. G1x)     ⇐ !, ∀x. peval (Gx) (G1x)
peval  (∃x. Gx)    G1            ⇐ !, peval (Gy) G1
peval  (D ⇒ G)     G1            ⇐ !, norm_form D D1, hyp D1 ⇒ peval G G1
peval  Ga          Ga            ⇐ stop? Ga.
peval  Ga          G             ⇐ hyp D, match_rule D (Ga ⇐ Gs), apply_rule? D, peval Gs G.

match_rule      D          D1  ⇐ match_rule_and D D1
match_rule_and  (D1 , D2)  D   ⇐ !, (match_rule_and D1 D; match_rule_and D2 D).
match_rule_and  D          D1  ⇐ !, match_rule_pi D D1
match_rule_pi   (∀x. Dx)   D1  ⇐ !, match_rule_pi (Dx) D1.
match_rule_pi   D          D.

left?       G   ⇐ write G, write_string "Left branch? : ", read λq.q.
stop?       Ga  ⇐ write Ga, write_string "Stop? : ", read λq.q.
apply_rule? D   ⇐ write D, write_string "Apply? : ", read λq.q.
```

Figure 4.8: Interactive Partial Evaluator

4.7 Chunking *vs.* EBG

Yet another paradigm has been compared to EBG, this one coming out of Soar. "Soar is an attempt to build a general cognitive architecture combining general learning, problem solving, and memory capabilities" [112, p.561].[11] *Chunking* is the learning mechanism of the Soar architecture. Rosenbloom and Laird present the case that EBG is very similar to chunking [112]. This correspondence may be expressed as the following mapping between Soar and our logic programming formulation of EBG:

Chunking	EBG
goal	goal
problem state	training theory
problem operators	domain theory
chunks (new productions)	derived rules
problem solution	proof (logic programming trace)
backtraced & variablized production sequences	generalized proofs
whether productions for a predicate exist	operationality criteria

[11] For an introduction to Soar and chunking, see Laird, Rosenbloom, & Newell [74, 75].

Soar solves problems at two-levels: Initially, learned rules of the production system are employed. When this strategy leads to an impasse — *i.e.*, no production solves the problem, Soar reverts to the operators in the problem space (theory). Upon solving the goal within the 'ground' problem space, Soar backtracks and 'variablizes' (*i.e.*, generalizes) the problem solving trace. The result is a chunk, or derived production rule, which, when added to the production system, potentially speeds future problem solving. Soar's subgoals are assumed to be 'operational' if there are pre-existing productions relevant to that subgoal.

Hopefully, this gross simplification of the Soar architecture has given unfamiliar readers a rudimentary understanding of chunking. We include this discussion as it will be relevant to future comparisons (§6.3), and further as it raises one problem with the application of the existing Soar/EBG correspondence to our formulation of EBG — namely that the mapping of $\mathcal{T}$ to a state space is not satisfying. The problem is that $\lambda^{\square}$Prolog provides a more expressive training theory than the training instance of typical EBG formulations. It is unclear how $\lambda^{\square}$Prolog's enhanced ability to abstract proof steps maps onto Soar.

4.8 Inductive Generalization

Within this section, we further discuss, for the interested reader, some relevant inductive methods of generalization. As this discussion does not bear on future chapters, it may be skipped by those so inclined.

Anti-unification. Perhaps the simplest method of generalizing an expression is to abstract a particular subexpression with a variable. This establishes a partial order of *instance* — one expression is an instance of another if the former may be derived from the latter by substituting terms for variables. The related process of unification, as discussed in §3.2, determines whether there exists a substitution deriving a common instance from two or more expressions.

We may also define the duals of these notions: *anti-instance* — one expression is an anti-instance of another if the former may be derived from the latter by abstracting subexpressions with variables, and *anti-unification* — the process of deriving a common anti-instance of two or more expressions.[12] In defining anti-unification, we would like to introduce the concept of a *least general anti-unifier* (LGAU), analogous to the most general unifier (MGU) of unification.[13] However, just as higher-order unification does not admit MGU's, *higher-order anti-unification* does not admit LGAU's:[14] consider that the anti-unification of $\lambda x.\mathbf{f}(\mathbf{g}(x))$ and $\lambda x.\mathbf{f}(\mathbf{h}(x))$ yields $\lambda x.\mathbf{f}(F(\mathbf{g}(x), \mathbf{h}(x)))$ and $\lambda x.F(\mathbf{f}(\mathbf{g}(x)), \mathbf{f}(\mathbf{h}(x)))$, again neither of

[12]Anti-unification was introduced independently by Plotkin [107] and Reynolds [109]. Dietzen & Scherlis tentatively discuss anti-unification in program development under the name 'least general generalization' [33]. For another treatment of first-order unification and anti-unification, see Lassez, Maher & Marriott [76].

[13]Somewhat more formally, the instance relation forms a lattice on the expression language in which MGU is the *meet* operator, and LGAU is the *join* [67]. (For an accessible introduction to lattices, see Stoy [125].)

[14]The higher-order case, then, does *not* admit a lattice since meets and joins are not unique.

which is an instance of the other. Higher-order anti-unification is further developed within Pfenning [105].

As it employs multiple examples, anti-unification is an inductive or similarity-based method of generalization. While anti-unification itself only handles positive instances, the technique may be extended to incorporate negative ones as well, which leads to Mitchell's version spaces [93]. Version spaces potentially admit a higher-order treatment that is founded upon higher-order unification and anti-unification. Indeed, this is an attractive area for research, although higher-order version spaces are complicated by the lack of MGU's and LGAU's.

Generalizing logic programs. Anti-unification is strictly a syntactic technique, but there are, of course, 'deeper' methods of abstracting expressions. In the case of logic programming, one goal might be considered more general than another if it is true for a greater range of instantiations to its variables. For instance, we could drop conditions from a conjunction: (**long** x, **wide** x) becomes **long** x. Or we might add conditions to a disjunction: replacing **long** x with (**long** x; **wide** x).[15] As we have remarked, although this disjunctive expression is more 'general' (in that it is less specific), it is not any more 'abstract' (in that it is not any less detailed). In fact, this is in some sense a trivial generalization, since arbitrarily complex concepts may be described as disjunctive sequences (perhaps infinite) of specific expressions. This same problem surfaces in higher-order anti-unification: the instances **long table** and **wide table** could be trivially anti-unified to f (**long table**) (**wide table**), or more succinctly to (f **long wide**) **table**. This result is considered 'disjunctive' because, in the simplest case, f is instantiated to project *either* its first *or* its second argument.

Each of the above techniques of generalizing logic programs — abstracting expressions with variables, dropping conjuncts, and adding disjuncts — can be exploited within inductive paradigms, thereby leading to approaches such as Vere's treatment of first-order predicate calculus [130] and Buntine's generalized subsumption over first-order Horn clauses [12]. The extension of these paradigms to encompass λProlog is yet another interesting topic for study.

The application of inductive techniques. While not considered herein, similarity-based techniques of generalization are, nevertheless, relevant to the overall vision of tools for design-based problem solvers articulated in Chapter 2. For one, this is because inductive reasoning is required for analogical problem solving; that is, similarity-based generalization is a means by which an analogical correspondence can be established between a solved and unsolved problem. The combination of this correspondence and the known solution serve as a guide in the construction of a new derivation. Analogical problem solving in this manner remains safe so long as the resulting derivation (for our domain, a logic programming proof) can be 'replayed' to establish validity.

[15] For some further possibilities of generalization see Dietterich, *et al.* [30, pp.365–368].

Chapter 5

The Extension of Logic Programs

In the previous chapter, we defined and illustrated the process of EBG. The resulting explanation-based generalizations, however, are of little use unless we also provide a means for *assimilation* — that is, for augmenting the existing logic program $\mathcal{P}$ with new rules. We define *learning*, then, to be the combination of generalization and assimilation. But rather than reserving the term 'assimilation' for the addition of clauses derived through EBG, we instead use it to refer to the extension of $\mathcal{P}$ with arbitrary D-forms.

The design of an appropriate assimilation mechanism for λProlog is complicated by our desire that it have the following characteristics:

- *Semantic simplicity.* The designers of λProlog took care to cultivate semantic elegance within the language. Therefore, we require that the language primitives we introduce for controlling generalization and assimilation continue in the λProlog 'spirit'. Primarily, this means that such mechanisms admit a *declarative* semantics, which should permit guarantees such as "this additional assumption already follows from the program." By 'declarative' we mean that the effects of a construct be readily scrutinized; *i.e.*, that the construct have a straight-forward definition which is easy to reason about. Moreover, this declarative characterization should exist apart from any particular operational interpretation (language implementation). (The distinction between declarative and operational descriptions will be further developed as we progress through this chapter.)

- *Programmability.* Any proposed assimilation mechanism should truly be 'programmable', since we do not believe that automatic (*i.e.*, uncontrolled) assimilation of derived facts or generalizations is either practical or desirable. Under a naive approach to learning, the underlying problem solving architecture produces and assimilates generalizations in the course of solving each query (at least when learning is 'switched on'). Except for the fact that explanation-based generalizations represent abstractions of computation, this approach to learning is analogous to following the solution of every Prolog query G with **assert** G. The resulting proliferation of clauses leads to increased matching overhead, and then perhaps to impaired rather than improved performance. Consequently, it generally becomes necessary that assimilation under such a paradigm either

be selective (*e.g.*, via some performance analysis) or involve the forgetting of those derived rules only infrequently used.[1]

Confining generalization and assimilation to the underlying architecture is problematic:

- To avoid the additional cost of producing generalizations and that of matching against a proliferation of rules in the program-base, generalization should be only selectively enabled (in our view, by the programmer, through the programming language).
- The results of EBG should be subject to modification by the program before assimilation: for reasons we shall illustrate, it is often desirable for the assumed clause to differ from the derived rule. Typically, this variation may be expressed as a simple forward reasoning step.

Thus, we advocate an architecture in which generalization and assimilation are realized *through language features* rather than *as aspects of the underlying system* (and therefore inaccessible to programmer control). The resulting language, $\lambda^{\square}_{G}$Prolog, is intended to serve as an effective platform for programming higher-order applications relying upon explanation-based learning.

In developing a more logical approach to explanation-based learning, we recognized the need for more limited forms of generalization within λProlog. The limited generalization to which we are referring is that of universally quantifying, or *universally generalizing*, existing free variables. Within Prolog this is accomplished by **assert**, in that **assert** (**p** x) implicitly universally quantifies x, effectively adding the clause $\forall x.\ \mathbf{p}\ x$ to the program.

The logic programming predicates **assert**, **retract**, **call**, **univ**, and **var** may be characterized as 'meta-logical', because they are concerned more with the manipulation of logic programs, including the currently running program itself, than with the logic of the language.[2] That is, they function at a fundamentally different level — the meta-level, and therefore, are typically defined *only* operationally (*e.g.*, within an interpreter), and apart from the logical foundation of language. This leads to problems in analysis and compilation (see [79], for example).

In this chapter we provide a logical foundation to a class of applications previously requiring **assert**. To that end, we propose the **rule** construct, which introduces a limited element of forward reasoning into λProlog. **rule** allows us to program in a natural and declarative way many meta-programming applications — *e.g.*, memoization, partial evaluation combined with reflection, and resolution — that heretofore relied upon extra-logical features.

Later, we develop an analogous construct, **rule_ebg**, which also extends a program by one of its consequences. The difference is in the consequence to be assumed: **rule**'s assumption is

---

[1] For a discussion of these issues see Prieditis & Mostow [108, pp.496–497], Minton [91], and Donat & Wallen [37].

[2] **call** G causes G to be solved; the λProlog equivalent is simply G. **univ** provides for the destructuring of Prolog terms into a *functor* (predicate) and arguments; within λProlog, higher-order unification addresses this task. **var** M succeeds if M is a logical variable and fails otherwise; there is no λProlog analog.

derived through universal generalization, while **rule_ebg**'s is the result of explanation-based generalization.

Although both of these constructs are proposed and applied in the framework of λProlog, the underlying ideas are general, and thus relevant to other logic programming languages.

Other work. We briefly mention three efforts concerned with establishing a formal account of meta-level logic programming constructs: As discussed in Chapter 3, λProlog [100] uses a fragment of higher-order intuitionistic logic, and gives a logical foundation for **call** (through higher-order predicates) and some uses of **assert** (through embedded implication). HiLog [17] uses an even narrower fragment of higher-order logic, and can give a declarative account for many uses of **univ** and **call**. The language Gödel [14] follows a different approach by explicitly separating the meta-level from the object-level.

5.1 Existing Approaches to Extending Logic Programs

5.1.1 Prolog's "assert"

Prolog permits the modification of the current logic program through the primitives **assert** and **retract**: **assert** D adds clause D to the program, while **retract** D removes D.[3] The following list characterizes the more frequent applications of **assert** and **retract** in Prolog:

- *Memoization* — To avoid the re-computation of previously solved goals, derived results are memoized, or cached.[4] Herein, the programmer must insure that **assert** is only applied to goals deductively following from the original program $\mathcal{P}$. We call this a *conservative extension* of $\mathcal{P}$. For example, the following definition of the Fibonacci function will not recompute values (unless it backtracks after an initial solution):

$$\begin{array}{llll} \textbf{fib} & 0 & 1. & \\ \textbf{fib} & 1 & 1. & \\ \textbf{fib} & m & n & \Leftarrow \quad m > 1,\ m_1 \textbf{ is } m-1,\ m_2 \textbf{ is } m-2, \\ & & & \quad \textbf{fib}\ m_2\ n_2,\ \textbf{asserta}\ (\textbf{fib}\ m_2\ n_2), \\ & & & \quad \textbf{fib}\ m_1\ n_1,\ \textbf{asserta}\ (\textbf{fib}\ m_1\ n_1),\ n \textbf{ is } n_1+n_2. \end{array}$$

  Memoization is a rudimentary form of *forward reasoning* — which we define to be any paradigm in which a knowledge-base (in this case, a logic program) grows by computing and assimilating facts that follow deductively. Although individual goals are derived through the standard backchaining of Prolog, their assimilation represents a forward reasoning step.

---

[3] Prolog implementations typically offer both **asserta** and **assertz**: the former effectively adds the clause to the beginning of the program, while the latter does so at the end. For purposes of general discussion, our use of **assert** encompasses both constructs.

[4] There is an inherent tradeoff in the application of memoization: the overhead of matching against a proliferating set of program clauses can result in deteriorating rather than an improved performance.

- *Interaction memoization* — This is an alternative application of memoization in which a program queries the user for assistance, and then records the result of the interaction with **assert**. Such extensions to the program are generally not conservative. See Rowe [113, pp.126–127] for an example.

- *Program reflection* — Reflection is the mapping of the data structure representing a program into an executable version of that program. (Its inverse — the mapping of an executable into data structure for scrutiny or manipulation — is *reification.*) The need for reflection arises when one wants to run a program constructed by another program. Reflection allows the derived program to be executed directly, in that way avoiding the inefficiency and complexity of interpreting a program datatype.

  The results of partial evaluation (PE) represent one important application of reflection. In the context of logic programming, partial evaluation consists of deriving a sufficient condition G for a particular query E[5]; that is, PE produces a *specialization* of the logic program $\mathcal{P}$ that captures the computation leading from E to G. Through use of the resulting derived rule $E \Leftarrow G$, we avoid re-doing the intervening computation. In §4.6, we introduced an interactive partial evaluator **peval** for λProlog. Rules derived through **peval** could be assimilated with **assert**, as in the top-level predicate **peval_top**:

  $$\textbf{peval\_top}\ E \ \Leftarrow\ \textbf{peval}\ E\ G,\ \textbf{asserta}\ (E \Leftarrow G).$$

  In §5.3.1 we show how reflection can be achieved declaratively with our proposed **rule** construct.

- *Retaining information across a failure* — The assumptions made by **assert** extend beyond a failure; that is, backtracking does not **retract** asserted clauses. This leads to the using of **assert** as a means to communicate 'across' a failure, which we illustrate through the coding of a **bagof** predicate. **bagof** produces a list L of every instance that satisfies a given single-argument predicate P. The following implementation of **bagof** exploits logic programming's backtracking search to iterate over potential values for P's argument x. **assert** and **retract** are used to maintain the intermediate values of this iteration.[6]

  $$\begin{array}{lll} \textbf{bagof} & P\ L & \Leftarrow\ \textbf{asserta (temp nil), fail.} \\ \textbf{bagof} & P\ L & \Leftarrow\ Px,\ \textbf{temp}\ K,\ \textbf{retract (temp}\ K\textbf{)}, \\ & & \quad\ \textbf{asserta (temp}\ (x :: K)\textbf{), fail.} \\ \textbf{bagof} & P\ L & \Leftarrow\ \textbf{temp}\ L,\ \textbf{retract (temp}\ L\textbf{).} \end{array}$$

  The first clause initializes **temp** — an accumulator for L. The second 'iterates' (via backtracking search) over values of x that satisfy P, storing them within K. When no more such x's can be found, the 'bag' is returned by the third clause.

---

[5] As introduced within Chapter 4, the class E stands for λProlog logical expressions that are both clauses and goals — *i.e.*, $E = D \cap G$.

[6] This encoding of **bagof** is a higher-order version of Rowe's [113, pp.236–238].

- *Mutable data* — **assert** and **retract** additionally support the side-effecting of global data structure. For instance, a breadth-first graph search may be implemented by updating a **fringe** clause that contains the set of vertices currently on the *fringe* of the search [113, p.234]. Similarly, **assert** could be employed to side-effect the graph itself for computing, say, its transitive closure.

- *Search control* — **assert** and **retract** can be used to supersede Prolog's normal depth-first search by hiding, reordering, or revealing clauses, or instead by setting global 'variables' to perform the same function. For example, Rowe describes a 'focus-of-attention' forward reasoner which recalls facts (via a **fact** predicate) for forward-chaining and then shifts them to 'used' status (by retracting **fact** and asserting **usedfact**) [113, pp.137–140].

Of course, **assert** (in combination with **retract**) also supports more general instances of self-modifying code, and is largely accepted as an important and necessary feature of logic programming languages. The principle drawback of **assert**, however, is that it has no accessible declarative meaning. Consequently, work on the semantics of logic programs typically ignores it: consider that there is no straightforward means for incorporating **assert/retract** within inference systems such as that defined in §3.7. And as a result, Prolog implementations behave inconsistently: Lindholm & O'Keefe [79, p.22] offer the example

p $\Leftarrow$ **assertz p, fail.**
p $\Leftarrow$ **fail.**

Whether $? - \mathbf{p}$ will succeed or fail depends upon the semantics of **assert**: Given a goal to solve **p**, should the set of relevant clauses be determined once for **p**'s solution, or should it instead be dynamically adjusted (following changes in logic program itself) in the course of solving **p**. Under the former interpretation, $? - \mathbf{p}$ fails, while under the latter it succeeds.

Given the more dynamic approach, there are still potential inconsistencies in the availability of additions to the logic program. The alternative example

q $\Leftarrow$ **fail.**
q $\Leftarrow$ **assertz q, fail.**

behaves differently in some Prolog implementations: the Warren Abstract Machine [132] — an abstract interpreter forming the basis of several Prolog implementations — succeeds on $? - \mathbf{p}$ and fails on $? - \mathbf{q}$.

For this and other reasons, λProlog does not include **assert**, although some of **assert**'s functionality is subsumed by another construct — embedded implication. However, as we shall illustrate, embedded implication is neither powerful enough to support the above applications of **assert**, nor for that matter, to support the assimilation of explanation-based generalizations. This lead us to explore the possibility of making logically motivated extensions to λProlog that address these deficiencies. In particular, we focus upon those uses of **assert** above that involve memoization and reflection. The other illustrations of **assert** are frequently stylistically questionable, and can often be reformulated without **assert** in a manner no more complex and no less efficient. In any case, **rule** is *not* intended to subsume the functionality of **assert**, but rather to provide a more declarative alternative for some subset of its uses.

5.1.2 Embedded Implication

It has been argued in the literature [83, 47, 9] that implication (with its intuitionistic meaning) can, in many situations, be used in place of **assert**, and can also be given a simple declarative semantics. The operational reading of embedded implication is that when solving the goal $D \Rightarrow G$, assume D while solving G. Thus without any program, the query

$$?- \ \mathbf{p}\ 1 \Rightarrow \mathbf{p}\ x.$$

succeeds with the answer $x = 1$. The assumption of an implication is in effect exactly while solving the consequent, and hence

$$?- \ (\mathbf{p}\ 1 \Rightarrow \mathbf{p}\ x),\ \mathbf{p}\ y.$$

will fail, though

$$?- \ ((\mathbf{p}\ 1,\ \mathbf{p}\ 2) \Rightarrow \mathbf{p}\ x),\ x = 2.$$

succeeds after some backtracking.

Implication is of particular importance when we wish to make an assumption for a particular computation and then 'forget' it. Consider a reformulation of **peval_top**:

$$\mathbf{peval\_top}\ E\ K \ \Leftarrow \ \mathbf{peval}\ E\ G,\ (E \Leftarrow G) \Rightarrow K.$$

The revised **peval_top** takes two arguments: the goal E to be partially evaluated, and a second goal K representing the context, or scope, for which the assumption $E \Leftarrow G$ will be valid. ('K' is for 'continuation', which is developed below.) The rationale behind **peval_top** is that the client has some computation (captured in K) for which a particular specialization of the program $E \Leftarrow G$ is applicable, yet he does not desire to make that optimization permanent (since, perhaps, it impairs performance in the general case).
At first, it might appear that the following definition would behave identically:

$$\mathbf{peval\_top}\ E\ K \ \Leftarrow \ \mathbf{peval}\ E\ G,\ \mathbf{asserta}\ (E \Leftarrow G),\ K,\ \mathbf{retract}\ (E \Leftarrow G).$$

However, the above is *not* equivalent to the preceding version. Suppose that the computation associated with K also makes extensions to the logic program. Should one of these assumptions unify with $E \Leftarrow G$, $\mathcal{P}$ could be left in an inconsistent state. Such potentially conflicting side-effects illustrate the difficulty in reasoning about programs that use **assert**; that is, they illustrate the non-declarative nature of those programs.

In fact, **assert** and **retract** are sufficient to encode implication in general, subject to limitations discussed below. The following uses **assert** and **retract** to establish the appropriate assumption, even in the face of backtracking:[7]

$$\begin{array}{lll}(D \Rightarrow G) & \Leftarrow & (\mathbf{asserta}\ D;\ (\mathbf{retract}\ D,\ \mathbf{fail})),\\ & & G,\\ & & (\mathbf{retract}\ D;\ (\mathbf{asserta}\ D,\ \mathbf{fail}))\end{array}$$

[7]This formulation is due to Stuart Shieber.

where backtracking over the first conjunct retracts D, and backtracking over the third re-asserts D (for G's solution). Of course, the above implementation does not address conflicting side-effects, universal generalization (§5.1.3), or the precise scoping of the asserted clause (§5.1.4).

5.1.3 Universal Quantification in Assumptions

In a Horn logic, all assumptions are closed: whatever apparently free variables occur in a clause D are in fact universally quantified. Because of this, a Horn logic program cannot change during its execution (at least, not without the application of meta-logical predicates such as **assert**). As pointed out in Chapter 3, this is *not* the case for logics that include embedded implication: assumptions therein added to $\mathcal{P}$ may contain logic variables that are not copied when that clause is used. Instead, the program may actually change (through variable instantiation) in the course of solving G. Thus, we distinguish between the assumptions $\mathbf{p}\ x$ and $\forall x\ \mathbf{p}\ x$. This is no great inconvenience: a clause occurring at the top-level in a program (typically those read-in from a file) is still considered to be universally quantified over its free variables, but no such convention exists for embedded implications.

This points out a manner in which implication is less powerful than **assert**: the former's assumption is not universally generalized. For instance,

$$?-\ \mathbf{asserta}\ (\mathbf{p}\ x),\ \mathbf{p}\ 1,\ \mathbf{p}\ 2.$$

succeeds in Prolog, while

$$?-\ \mathbf{p}\ x \Rightarrow (\mathbf{p}\ 1,\ \mathbf{p}\ 2).$$

fails in λProlog: as one can see, there is no x such that $\mathbf{p}\ x$ implies both $\mathbf{p}\ 1$ and $\mathbf{p}\ 2$. Operationally, what happens is that resolving $\mathbf{p}\ 1$ with the assumption $\mathbf{p}\ x$ instantiates x to 1, and the now instantiated assumption $\mathbf{p}\ 1$ does not unify with the second subgoal $\mathbf{p}\ 2$. On the other hand, the following clearly succeeds:

$$?-\ (\forall x.\mathbf{p}\ x) \Rightarrow (\mathbf{p}\ 1,\ \mathbf{p}\ 2).$$

It should be remarked here that this behavior of embedded implication is not a design mistake, but has its applications, and, furthermore, is entailed by the desire to make only logically sound extensions to basic Horn logic (for a further discussion see [83]).

This limitation of implication does illustrate a problem within the preceding formulation of **peval_top**: a clause $E \Leftarrow G$ derived by partial evaluation and then assumed (via implication) can only be used with one substitution for its logical variables. We conclude that neither implication nor **assert** is the proper mechanism for the situation as described.

5.1.4 Goal Continuations

Implication is also restricted in that its precise scope can limit the exploitation of its assumptions. Consider another definition of Fibonacci which attempts to exploit implication for memoization:

$$\begin{array}{llll} \textbf{fib} & 0 & 1. & \\ \textbf{fib} & 1 & 1. & \\ \textbf{fib} & m & n & \Leftarrow \; m > 1,\ m_1 \textbf{ is } m-1,\ m_2 \textbf{ is } m-2, \\ & & & \quad \textbf{fib}\ m_2\ n_2,\ \textbf{fib}\ m_2\ n_2 \Rightarrow \textbf{fib}\ m_1\ n_1,\ n \textbf{ is } n_1+n_2. \end{array}$$

The problem here is that **fib**'s assumptions are not uniformly visible: consider a recursive trace of **fib** as a binary tree, where the computation of the nth Fibonacci F_n is reduced to that of F_{n-2} and F_{n-1}, with F_{n-2} known for the latter:

$$\begin{array}{ccccccc} & & & F_n & & & \\ & F_{n-2} & & & & F_{n-2} \Rightarrow F_{n-1} & \\ F_{n-4} & & F_{n-4} \Rightarrow F_{n-3} & & F_{n-3} & & F_{n-3} \Rightarrow F_{n-2} \end{array}$$

So in the recursive computation of F_{n-1}, the only pending assumption is that of F_{n-2}, and hence F_{n-3} must be re-derived. Thus, while the above performs considerably better than a similar program without implication, the original version using **assert** is substantially more efficient (linear).

This problem can effectively be circumvented by reformulating the program in *continuation-passing style* (CPS) [110], which the reader may have encountered in the context of functional programming. To realize CPS under λProlog, we add another argument K (a goal) to our predicate. This goal is intended to represent the remainder of the computation, and thus rather than returning control upon success, clauses invoke this 'goal continuation.' In this way, accumulated assumptions are made available to extended computations. The following formulation of **fib** makes use of CPS:

$$\begin{array}{llllll} \textbf{fib} & m & n & & \Leftarrow & \textbf{fib}_1\ m\ n\ \textbf{true}. \\ \textbf{fib}_1 & 0 & 1 & K & \Leftarrow & K. \\ \textbf{fib}_1 & 1 & 1 & K & \Leftarrow & K. \\ \textbf{fib}_1 & m & n & K & \Leftarrow & m > 1,\ m_1 \textbf{ is } m-1,\ m_2 \textbf{ is } m-2, \\ & & & & & \quad \textbf{fib}_1\ m_2\ n_2\ ((\forall K'.\, \textbf{fib}_1\ m_2\ n_2\ K' \Leftarrow K') \Rightarrow \\ & & & & & \qquad \textbf{fib}_1\ m_1\ n_1\ ((\forall K'.\, \textbf{fib}_1\ m_1\ n_1\ K' \Leftarrow K') \Rightarrow (n \textbf{ is } n_1+n_2, K))). \end{array}$$

While this illustration may be somewhat inscrutable to those not acclimated to CPS (higher-level notations would be helpful), the underlying intuition is not that difficult: K captures the computation necessary to solve a pending $\textbf{fib}_1$ calculation. In that regard, it acts as an accumulator for the pending subgoals of that computation. Within the last clause, $\textbf{fib}_1\ m_2\ n_2$ is computed, and then the program is extended with the assumption that for any K, $\textbf{fib}_1\ m_2\ n_2\ K$ reduces to K (since m_2 and n_2 have been instantiated to particular values). In this way, $\textbf{fib}_1\ m_2\ n_2$ need not be re-calculated by any computation nested within K.

Our motivation for this digression into CPS is that it is a powerful mechanism through which implication (and later our extension, **rule**) can be more fully exploited within λProlog.

5.1.5 Persistence of Assumptions

A further limitation of implication's well-defined scope is that there does not appear to be a means for making assumptions persistent. eLP (our λProlog implementation) circumvents this with no loss of elegance by allowing the programmer to initiate a new top-level interpretation via the special goal **top** (introduced by Pfenning), which recursively invokes a new λProlog listener, thereby effectively 'globally' extending the existing logic program with any pending assumptions, such as within $\mathbf{fib}_1\ m\ n\ \mathbf{top}$.

5.1.6 Summary

We have seen that **assert** and **retract** are insufficient to program implication, due to the lack of proper scoping and the possibility of conflicting side-effects. Conversely, we find that there are three aspects of **assert** which are difficult to model with embedded implication:

1. global accessibility of the asserted clause, although this can often be achieved using continuation passing style;
2. persistence of the asserted clause, which has been addressed in λProlog with the special predicate **top**; and
3. universal generalization of assumed clauses.

The last of the three, universal generalization, is the most problematic, because there is often no way to program it short of completely reformulating the data representation.[8] It is also universal generalization which is addressed by our proposed **rule** construct. Since **rule** resembles implication in that its assumptions are always given a limited scope, the techniques employed in (1) and (2) will continue to be relevant for programming with the new construct.

5.2 Lemma

We seek to address, in a declarative manner, embedded implication's inadequacy with regard to universal generalization. To that end, in the section to follow we propose the **rule** construct. Before we introduce **rule**, however, we first attempt to motivate that extension with a less general counterpart, the **lemma** construct. **lemma**, which we later establish to be a special case of **rule**, brings to light many issues relevant to **rule**'s development.

[8] This is essentially the solution advocated by Burt *et al.* [14].

5.2.1 Prolog's "lemma"

The semantic inelegance of **assert** has lead to the consideration of other means by which logic programs may be extended. One such alternative is **lemma** as described by Sterling & Shapiro [124, p.181], which may be defined within Prolog as

$$\textbf{lemma}\ E \ \Leftarrow\ E,\ \textbf{asserta}\ E.$$

(Actually, Sterling & Shapiro's formulation is

$$\textbf{lemma}\ E \ \Leftarrow\ E,\ \textbf{asserta}\ (E \Leftarrow !).$$

which does not backtrack to other clauses if a lemma applies.)

lemma is more 'logical' than **assert** in that it only permits conservative extension — that is, added clauses necessarily follow from the theory described by the logic program. We may again reformulate Fibonacci as

$$\begin{array}{l}\textbf{fib}\ \ 0\ \ 1.\\ \textbf{fib}\ \ 1\ \ 1.\\ \textbf{fib}\ \ m\ \ n\ \ \Leftarrow\ \ m > 1,\ \ m_1\ \textbf{is}\ m-1,\ \ m_2\ \textbf{is}\ m-2,\\ \qquad\qquad\qquad\ \ \textbf{lemma}\ (\textbf{fib}\ m_2\ n_2),\ \ \textbf{lemma}\ (\textbf{fib}\ m_1\ n_1),\ \ n\ \textbf{is}\ n_1+n_2.\end{array}$$

5.2.2 A Scoped "lemma" Construct

Prolog's **lemma** takes a single argument — the goal E to be solved and then assumed. We will now define an analogous **lemma** within λProlog. The new **lemma** more resembles implication in that it gives its assumption a proper scope. λProlog's **lemma**, then, requires two arguments: (1) the goal E to be solved and assumed, and (2) a goal K representing the scope for which the assumption is valid. **lemma** $E\ K$ may be informally characterized as

$$\textbf{lemma}\ E\ K \ \Leftarrow\ E,\ (E' \Rightarrow K).$$

where E' is a universal generalization of E.

To illustrate an application of this scoped **lemma**, consider one last reformulation of **fib**:

$$\begin{array}{l}\textbf{fib}\ 0\ 1.\\ \textbf{fib}\ 1\ 1.\\ \textbf{fib}\ m\ n\ \Leftarrow\ m > 1,\ \ m_1\ \textbf{is}\ m-1,\ \ m_2\ \textbf{is}\ m-2,\\ \qquad\qquad\ \ \textbf{lemma}\ (\textbf{fib}\ m_2\ n_2)\\ \qquad\qquad\qquad\ \ (\textbf{lemma}\ (\textbf{fib}\ m_1\ n_1)\\ \qquad\qquad\qquad\qquad\ \ (n\ \textbf{is}\ n_1+n_2)).\end{array}$$

(This version suffers from precisely the same inefficiency as that relying upon embedded implication. In fact, the two are equivalent. See §5.1.4.) As with embedded implication, persistence can be achieved with **lemma** E **top**.

Yet **lemma** is in some ways more powerful than embedded implication. For example, new clauses may be derived through **lemma**: from the program

$$\begin{array}{lllll} \textbf{child\_of} & x & y & \Leftarrow & \textbf{parent}\ y\ x. \\ \textbf{parent} & x & y & \Leftarrow & \textbf{child\_of}\ y\ x. \\ \textbf{ancestor} & x & y & \Leftarrow & \textbf{parent}\ x\ y;\ (\textbf{parent}\ x\ z,\ \textbf{ancestor}\ z\ y). \end{array}$$

we may derive the general goal

$$?-\ \ \textbf{child\_of}\ x\ y\ \Rightarrow\ \textbf{ancestor}\ y\ x.$$

for arbitrary (uninstantiated) x and y. **lemma** affords the universal generalization of such variables: the goal

$$?-\ \ \textbf{lemma}\ (\textbf{child\_of}\ x\ y\ \Rightarrow\ \textbf{ancestor}\ y\ x)\ K.$$

will assume the derived clause

$$\forall x\ \forall y.\ \textbf{ancestor}\ y\ x\ \Leftarrow\ \textbf{child\_of}\ x\ y.$$

before attempting the solution of K. In this way, **lemma** supports the extension of the program with new universally quantified clauses that follow from that program. Implication alone cannot universally generalize x and y.

The operational reading of $\mathcal{P} \vdash \textbf{lemma}\ E\ K$ is

> Solve $\mathcal{P} \vdash E$. If this fails, backtrack. Otherwise, it succeeds with substitution θ. Let $\mathcal{Y}$ be the set of the free variables remaining in θE that do not appear free in $\theta\mathcal{P}$, and let $\forall\mathcal{Y}$ stand for the universal quantification of each y in $\mathcal{Y}$. Thus, $\forall\mathcal{Y}.\,\theta E$ is the universal generalization of θE over variables that do not occur in $\theta\mathcal{P}$.
>
> Next solve
>
> $$\{\forall\mathcal{Y}.\,\theta E\} \cup \theta\mathcal{P}\ \vdash\ \theta K$$
>
> If this succeeds with substitution ψ, then $\mathcal{P} \vdash_{\psi\theta} \textbf{lemma}\ E\ K$.

Declaratively, **lemma** E K is simply equivalent to (E, K), the conjunction of E and K. By the *declarative equivalence* of two goals G_1 and G_2, we mean that the given goals follow from the same logic programs and are satisfied by the same substitutions — *i.e.*, $\theta\mathcal{P} \vdash \theta G_1$ if and only if $\theta\mathcal{P} \vdash \theta G_2$.

An *operational reading*, on the other hand, considers the search behavior (*e.g.*, ordering of selections and backtracking) of a specific logic programming interpretation. As an example of the difference, consider that the Prolog goal **once**

$$\textbf{once}\ G \ \Leftarrow \ G, !.$$

is declaratively the same as G: each is satisfied by similar substitutions and logic programs. Operationally, however, the interpreter does not backtrack to find alternative solutions of **once** G.

The purpose of **lemma** is to affect termination and efficiency without affecting provability: **lemma** controls search by selectively expanding the program. This expansion is through the assumption $\forall\mathcal{Y}.\ \theta E$. The savings afforded by **lemma** is simply that rather than successively re-deriving and re-instantiating E, it is derived once and then universally generalized to $\forall\mathcal{Y}.\ \theta E$, so that the latter may be exploited in the solution of multiple goals. As θE is necessarily a consequence of $\mathcal{P}$, $\forall\mathcal{Y}.\ \theta E$ follows from universal generalization via the *discharging* of logic variables not free in $\mathcal{P}$. Without the forward reasoning step resulting in the assumption $\forall\mathcal{Y}.\ \theta E$, the solution of K could not even terminate. Even if $\mathcal{P} \Vdash K$ succeeds, the discovered proof is potentially much longer than that associated with $\{\forall\mathcal{Y}.\ \theta E\} \cup \theta\mathcal{P} \Vdash \theta K$ (as is the case with **fib**).

Of course, **lemma** *cannot* be effectively implemented as

$$\textbf{lemma}\ M\ K \ \Leftarrow \ (M,\ M \Rightarrow K).$$

While this is a correct implementation in that it is equivalent to the declarative reading (M, K), it does not realize the operational definition — *i.e.*, it does not universally generalize, and thus will offer the same performance as (M, K).

In fact, **lemma** cannot be programmed within the existing language, since λProlog affords no means by which to universally generalize free variables. The universal generalization step, which is required in the implementation of both **lemma** and **assert**, is problematic for languages with embedded implication. Consider

$$?-\ \ \mathbf{p}\ x \ \Rightarrow\ \textbf{lemma}\ (\mathbf{p}\ x)\ (\mathbf{p}\ 1, \mathbf{p}\ 2).$$

Since $\mathbf{p}\ x$ trivially follows from itself, the above might naively be expected to make the assumption $\forall x.\ \mathbf{p}\ x$. But this does not follow from the program, since its declarative counterpart

$$?-\ \ \mathbf{p}\ x \ \Rightarrow\ (\mathbf{p}\ x, (\mathbf{p}\ 1, \mathbf{p}\ 2)).$$

is not true and, in fact, fails in λProlog.

The problem of defining **assert** within λProlog is even more dramatic: consider that there is no obvious meaning for

$$?- \quad \exists x \, \forall y. \, \mathbf{q} \; x \; y \; \Rightarrow \; \mathbf{asserta} \; (\mathbf{r} \; x \; y).$$

How can x and y be meaningfully quantified within a globally asserted clause? The difficulty is that λProlog goals are solved within the scope of a particular set of local assumptions and variable bindings. They are not free to 'stand alone' as Horn clauses are. Our version of **lemma** *is* reconcilable with λProlog because it too is scoped: **lemma**'s assumption $\forall \mathcal{Y}. \, \theta E$ is only valid within the scope of the deriving context — that is, is for the solution of K. An unscoped **assert**, on the other hand, does not make sense for λProlog, because its assumptions are expected to persist beyond the extent of the defining context.

5.2.3 Formal Definition

The operational definition of **lemma** may be formalized within the following inference rule (a la §3.7):

$$\frac{\mathcal{P} \Vdash_{\theta} E \qquad \{\forall \mathcal{Y}. \, \theta E\} \cup \theta \mathcal{P} \; \vdash_{\psi} \; \theta K}{\mathcal{P} \vdash_{\psi\theta} \mathbf{lemma} \; E \; K} \qquad \text{where } \mathcal{Y} = \mathsf{free}(\theta E) - \mathsf{free}(\theta \mathcal{P}).$$

The preceding operational reading ensures that **lemma** will succeed only if its corresponding declarative interpretation $(E \, , \, K)$ is valid. This property, the *soundness* of **lemma**, may be proved by induction on the definition of the $\Vdash$ relation (§3.7):

Given $\mathcal{P} \vdash_{\psi\theta} \mathbf{lemma} \; E \; K$, we must show that $\mathcal{P} \vdash (E \, , \, K)$.
From the definition of **lemma**, $\mathcal{P} \Vdash_{\theta} E$, and then from the *ind.hyp.* $\theta \mathcal{P} \vdash \theta E$.
Let $\mathcal{Y} = \mathsf{free}(\theta E) - \mathsf{free}(\theta \mathcal{P})$.
Since $\theta \mathcal{P} \vdash \theta E$ and $\mathcal{Y} \cap \mathsf{free}(\theta \mathcal{P}) = \emptyset$,
it follows by *universal generalization* that $\theta \mathcal{P} \vdash (\forall \mathcal{Y}. \, \theta E)$. (1)
Also from the definition of **lemma**, $\{\forall \mathcal{Y}. \, \theta E\} \cup \theta \mathcal{P} \; \vdash_{\psi} \; \theta K$,
and hence by the *ind.hyp.* $\psi\{\forall \mathcal{Y}. \, \theta E\} \cup \psi\theta \mathcal{P} \; \vdash \; \psi\theta K$. (2)
By *cut-elimination*[9] over (1) and (2), $\psi\theta \mathcal{P} \vdash \psi\theta K$
Again from (1), $\psi\theta \mathcal{P} \vdash \psi\theta E$.
And thus $\psi\theta \mathcal{P} \; \vdash \; \psi\theta E, \psi\theta K$.

[9]The rule of *cut-elimination* for $\vdash$ states that from $\mathcal{P} \vdash A$ and $\mathcal{P} \cup A \vdash B$, we may conclude $\mathcal{P} \vdash B$.

5.2.4 Alternative Realizations (Optional)

In order to achieve a correct realization of **lemma**, it is necessary to suppress the universal generalization of variables free in $\mathcal{P}$. Through embedded implication, then, the program may contain (in a temporary assumption) a free occurrence of x, which invalidates **lemma**'s universal generalization over x. This can be remedied in two ways: we can either (1) not generalize over variables free in an assumption (as per **lemma**'s definition above), or instead (2) collect each of the assumptions containing variables free in E within an enabling precondition (subgoal) of the lemma we create. For example, consider

$$?- \ \mathbf{p}\ x \ \Rightarrow\ \mathbf{lemma}\ (\mathbf{p}\ x)\ (\mathbf{p}\ 1, \mathbf{p}\ 2).$$

For case (1) the derived assumption is $\mathbf{p}\ x$, and for case (2) $\forall x.\, \mathbf{p}\ x \Rightarrow \mathbf{p}\ x$, rather than the incorrect $\forall x.\, \mathbf{p}\ x$. As a slightly more complex illustration, given only the program

$$\mathbf{q}\ x\ y \ \Leftarrow\ \mathbf{p}\ x, \mathbf{p}\ y.$$

the query

$$?- \ \mathbf{p}\ x \ \Rightarrow\ \mathbf{lemma}\ (\mathbf{q}\ x\ y)\ K.$$

would make either the assumption (1) $\mathbf{q}\ x\ x$, or (2) $\forall y.\, \mathbf{q}\ y\ y \Leftarrow \mathbf{p}\ y$, rather than the incorrect $\forall x.\, \mathbf{q}\ x\ x$.

For our implementation of **lemma**, we chose the first solution, since it seems to be more frequently useful. In fact, for most situations, (2) reduces to (1) since the only means for deriving the additional subgoal associated with (2) will be precisely via the initial assumption: that is, to derive the precondition $\mathbf{p}\ y$ in the most recent example, presumably one would need the local assumption $\mathbf{p}\ x$.

Effectively determining which variables appear free in assumptions is a potentially thorny implementation issue: given that there are a large number of pending assumptions, the search required is not insignificant. Maintaining a list of such variables seems the natural approach. Within §9.2.1 we discuss our implementation and its limitations in this regard.

5.3 Rule

5.3.1 Example: Partial Evaluation

Let us now return to the **peval_top** example introduced in §5.1.1. Recall that the problem with

$$\mathbf{peval\_top}\ E\ K \ \Leftarrow\ \mathbf{peval}\ E\ G,\ (E \Leftarrow G) \Rightarrow K.$$

is that the free variables of $E \Leftarrow G$ are not universally generalized, thereby restricting the applicability of the assumption.

Since $E \Leftarrow G$, though true, is typically not itself derivable as a goal (it is the result of partial evaluation; not logic programming execution), the formulation

$$\textbf{peval\_top}\ E\ K \ \Leftarrow\ \textbf{peval}\ E\ G,\ \textbf{lemma}\ (E \Leftarrow G)\ K.$$

is not sufficient. Nor can we augment **lemma** with a local assumption such as

$$\textbf{peval\_top}\ E\ K \ \Leftarrow\ \textbf{peval}\ E\ G,\ (E \Leftarrow G) \Rightarrow \textbf{lemma}\ (E \Leftarrow G)\ K.$$

because of the aforementioned restrictions placed upon variables free in assumptions. As one last attempt, consider

$$\begin{aligned}\textbf{peval\_top}\ E\ K \ \Leftarrow\ \textbf{lemma}\ \ &(\textbf{peval}\ E\ G)\\ &(\forall E\, \forall G.\, \textbf{peval}\ E\ G \Rightarrow (E \Leftarrow G)) \Rightarrow K).\end{aligned}$$

The reader might hope that this formulation would perform the partial evaluation, universally generalize the result, and then allow the result to be exploited through one additional embedded implication $\forall E\, \forall G.\, \textbf{peval}\ E\ G \Rightarrow (E \Leftarrow G)$. The problem is that this implication does not make any sense under logic programming's backtracking search paradigm: as discussed below, this clause has a variable head, and is hence applicable to any goal whatsoever.

Operationally, what we would like to achieve for $\vdash \textbf{peval\_top}\ E\ K$ is

1. Solve $\vdash \textbf{peval}\ E\ G$. If this succeeds with a substitution θ, let $\mathcal{Y}$ be the logic variables contained in θE and θG that do not occur free in any current assumption.
2. Assume (*i.e.*, *reflect*) $\forall \mathcal{Y}.\, \theta E \Leftarrow \theta G$ while solving θK; that is, solve

$$\{\forall \mathcal{Y}.\, \theta E \Leftarrow \theta G\} \cup \theta \mathcal{P} \ \vdash\ \theta K.$$

Why is this a sound way of establishing θK? We need to make three crucial observations:

1. Since we quantify only over those variables which are not free in any current assumption, we know that $\forall \mathcal{Y}.\, \textbf{peval}\ \theta E\ \theta G$ is a logical consequence of the program for **peval** (because of the logical rule of universal generalization).
2. The programmer knows that if $\textbf{peval}\ E\ G$, then $E \Leftarrow G$ is a valid clause to add to the program (assuming **peval** has been implemented correctly). This is expressed declaratively within the aforementioned clause

$$\forall E\, \forall G.\, \textbf{peval}\ E\ G \ \Rightarrow\ (E \Leftarrow G).$$

3. From a simple forward reasoning step, we conclude that $\forall \mathcal{Y}.\, \theta E \Leftarrow \theta G$ is true, and hence can be safely assumed before solving θK.

Trying to abstract from this particular example, we can see that we need two pieces of information in order to carry out the operations described above: the original goal to be solved — **peval** $E\ G$, and the general rule establishing the connection between this goal and the assumption we would like to make — $\forall E\, \forall G.$ **peval** $E\ G \Rightarrow (E \Leftarrow G)$. In order to properly scope assumptions, we also need to pass a goal continuation K as an argument. This line of reasoning is embodied within our new construct **rule**. For **peval_top**, the **rule** invocation is

$$\begin{aligned}\textbf{peval\_top}\ E\ K \Leftarrow\ & \textbf{rule}\ (\textbf{peval}\ E\ G) \\ & (\forall E\, \forall G.\ \textbf{peval}\ E\ G \Rightarrow (E \Leftarrow G)) \\ & K.\end{aligned}$$

The forward reasoning supported by **rule** takes the form of a single forward-chaining step (specified by an implication) that is under the tight control of the programmer. Intuitively, that step is to solve the left-hand side of the implication, and, upon success, assume the right. Such a step in the forward direction is generally incompatible with the backchaining of the logic programming paradigm: consider if we were to include $\forall E\, \forall G.$ **peval** $E\ G \Rightarrow (E \Leftarrow G)$ within $\mathcal{P}$, it would be applicable to any implicational G-form. (In fact, after conversion to normal-form, the given clause is applicable to any goal whatsoever.) Thus, this forward step is of little practical value outside of the **rule** context.

5.3.2 The "rule" Construct

The general form of **rule** is

$$\textbf{rule}\ G\ (\forall \mathcal{X}.\, G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ K$$

for goals G, $G_{\mathcal{X}}$, K, and clause $D_{\mathcal{X}}$, where $\mathcal{X}$ is a (perhaps empty) subset of the variables free in $D_{\mathcal{X}}$ or $G_{\mathcal{X}}$. To simplify the discussion, we assume that the variables in $\mathcal{X}$ do not occur elsewhere.

Operational reading. The operational interpretation of

$$\mathcal{P} \vdash \textbf{rule}\ G\ (\forall \mathcal{X}.\, G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ K$$

is as follows:

1. Find a minimal substitution $\sigma_{\mathcal{X}}$[10] such that $\mathsf{dom}(\sigma_{\mathcal{X}}) \subseteq \mathcal{X}$ and $\sigma_{\mathcal{X}} G_{\mathcal{X}} = G$. The existence of $\sigma_{\mathcal{X}}$ guarantees that the forward-chaining step is applicable. Should $\sigma_{\mathcal{X}}$ not exist, fail and issue a diagnostic message.

[10]Recall from §3.7 that a given substitution σ is *minimal*, or *most general*, with respect to a particular set of conditions, if θ satisfies those conditions, and if for any other substitution ψ also satisfying those conditions, ψ is a instance of θ.

2. Solve $\mathcal{P} \vdash G$. If this fails, fail. Otherwise, it succeeds with some substitution θ.

3. Let $\mathcal{Y} = \mathsf{free}(\theta G) - \mathsf{free}(\theta \mathcal{P})$.

4. Let $\mathcal{X}' = \mathcal{X} - \mathsf{dom}(\sigma_{\mathcal{X}})$.

5. Solve

$$\{\forall \mathcal{X}' \forall \mathcal{Y}. \theta \sigma_{\mathcal{X}} D_{\mathcal{X}}\} \cup \theta \mathcal{P} \vdash \theta K$$

(Relying upon λProlog unification rather than explicit substitution, the preceding operational description may alternatively be codified as

1. Create new logical variables $\hat{\mathcal{X}}$ for each universal variable in $\mathcal{X}$, and then substitute $\hat{\mathcal{X}}$ for $\mathcal{X}$ in $G_{\mathcal{X}}$ and $D_{\mathcal{X}}$, yielding $G_{\hat{\mathcal{X}}}$ and $D_{\hat{\mathcal{X}}}$.

2. Unify G and $G_{\hat{\mathcal{X}}}$.

3. Solve G.

4. Let $\mathcal{Y} = \mathsf{free}(G) - \mathsf{free}(\mathcal{P})$

5. Solve $(\forall \hat{\mathcal{X}}. \forall \mathcal{Y}. D_{\hat{\mathcal{X}}}) \Rightarrow K$.

$\forall \hat{\mathcal{X}}$ is the correct quantification, since those variables of $\hat{\mathcal{X}}$ which **rule** has instantiated no longer appear free in $D_{\hat{\mathcal{X}}}$.)

Declarative reading. The proper declarative interpretation for

$$\textbf{rule}\ G\ (\forall \mathcal{X}. G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ K$$

is simply

$$G,\ \ (\forall \mathcal{X}. G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}}) \Rightarrow K$$

which, like the reading for **lemma**, makes no mention of universal generalization whatsoever.

The difference between the operational and declarative readings illustrate the savings provided by **rule**: *Under the declarative interpretation, multiple instances of the same general goal* G must be solved in order to establish instances of G's consequent D. Operationally, however, we need solve G only once, universally generalize, and then assume the universal closure of its consequent D.

In fact, the reason for explicitly including the $\forall \mathcal{X}$, rather than just allowing the variables to be free, is that it is required by the declarative interpretation: since it may be necessary to repeat the application of the step $\forall \mathcal{X}. G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}}$, $\mathcal{X}$ must contain all variables that may be reinstantiated during such successive applications.

Note that $\forall\mathcal{X}.\, G_\mathcal{X} \Rightarrow D_\mathcal{X}$, though true, is never used in the backward-chaining search of the interpreter; only the result of the forward step is assumed. This is essential as the clause one typically uses for this forward-chaining step is often hopelessly inefficient, or else quickly leads to non-termination if used in the reverse direction. We have already given as an example the step in the definition of **peval_top**.

That it is the free variables of G, rather than those of D, which are universally generalized is essential for correctness: consider

$?-$ **rule true (true $\Rightarrow$ p x) (p 1, p 2).**

which fails, as established be the invalidity of the declarative reading:

$?-$ **true, (true $\Rightarrow$ p x) $\Rightarrow$ (p 1, p 2).**

The following variation, on the other hand, should (and does) succeed:

$?-$ **rule true ($\forall x$. true $\Rightarrow$ p x) (p 1, p 2).**

and, as we would expect, its declarative reading behaves similarly:

$?-$ **true, ($\forall x$. true $\Rightarrow$ p x) $\Rightarrow$ (p 1, p 2).**

5.3.3 Formal Definition

The above operational definition is formalized by the inference rule

$$\frac{\mathcal{P} \Vdash_\theta G \qquad \sigma_\mathcal{X} G_\mathcal{X} = G \qquad \{\forall\mathcal{X}'\,\forall\mathcal{Y}.\, \theta\sigma_\mathcal{X} D_\mathcal{X}\} \cup \theta\mathcal{P} \vdash_\psi \theta K}{\mathcal{P} \vdash_{\psi\theta} \textbf{rule}\ G\ (\forall\mathcal{X}.\, G_\mathcal{X} \Rightarrow D_\mathcal{X})\ K}$$

where $\mathsf{dom}(\sigma_\mathcal{X}) \subseteq \mathcal{X}$,
$\sigma_\mathcal{X}$ is minimal,[11]
$\mathcal{X}' = \mathcal{X} - \mathsf{dom}(\sigma_\mathcal{X})$, and
$\mathcal{Y} = \mathsf{free}(\theta G) - \mathsf{free}(\theta\mathcal{P})$.

As in the case of **lemma**, the above operational definition ensures that **rule** will succeed only if its corresponding declarative interpretation is valid. This property, the *soundness* of **rule**, is also proved by induction over the $\Vdash$ relation (§3.7):

Given $\mathcal{P} \vdash_{\psi\theta}$ **rule** $G\ (\forall\mathcal{X}.\, G_\mathcal{X} \Rightarrow D_\mathcal{X})\ K$,
  we must show that $\mathcal{P} \vdash G$ and $\{\forall\mathcal{X}.\, G_\mathcal{X} \Rightarrow D_\mathcal{X}\} \cup \mathcal{P} \vdash K$.
From the definition of **rule**, $\mathcal{P} \vdash_\theta G$, and then from the *ind.hyp.*, $\theta\mathcal{P} \vdash \theta G$.
Again from **rule**'s definition, there exists $\sigma_\mathcal{X}$ such that $\sigma_\mathcal{X} G_\mathcal{X} = G$,
  where $\mathsf{dom}(\sigma_\mathcal{X}) \subseteq \mathcal{X}$ and $\sigma_\mathcal{X}$ is minimal.
Thus $\theta\mathcal{P} \vdash \theta\sigma_\mathcal{X} G_\mathcal{X}$.

Let $\mathcal{Y} = \mathsf{free}(\theta G) - \mathsf{free}(\theta \mathcal{P})$, and let $\mathcal{X}' = \mathcal{X} - \mathsf{dom}(\sigma_\mathcal{X})$.
By *universal generalization*, $\theta\mathcal{P} \vdash \forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_\mathcal{X}G_\mathcal{X}$.
By *weakening*, $\theta\{\forall\mathcal{X}.\,G_\mathcal{X} \Rightarrow D_\mathcal{X}\} \cup \theta\mathcal{P} \vdash \forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_\mathcal{X}G_\mathcal{X}$.
By *universal instantiation* (via $\sigma_\mathcal{X}$) and $\wedge$-*introduction*, it follows that
$\theta\{\forall\mathcal{X}.\,G_\mathcal{X} \Rightarrow D_\mathcal{X}\} \cup \theta\mathcal{P} \vdash \forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_\mathcal{X}G_\mathcal{X},\ \theta\sigma_\mathcal{X}(G_\mathcal{X} \Rightarrow D_\mathcal{X})$.
And then by *distributivity* of substitution,
$\theta\{\forall\mathcal{X}.\,G_\mathcal{X} \Rightarrow D_\mathcal{X}\} \cup \theta\mathcal{P} \vdash \forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_\mathcal{X}G_\mathcal{X},\ \theta\sigma_\mathcal{X}G_\mathcal{X} \Rightarrow \theta\sigma_\mathcal{X}D_\mathcal{X}$
By *modus ponens* (rule application) and by induction over the preceding steps,
it then follows that $\theta\{\forall\mathcal{X}.\,G_\mathcal{X} \Rightarrow D_\mathcal{X}\} \cup \theta\mathcal{P} \vdash \forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_\mathcal{X}D_\mathcal{X}$. (1)
Also by the definition of **rule**, $\{\forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_\mathcal{X}D_\mathcal{X}\} \cup \theta\mathcal{P} \Vdash_\psi \theta K$,
and hence by the *ind.hyp.* $\psi\{\forall\mathcal{X}'\,\forall\mathcal{Y}.\,\theta\sigma_\mathcal{X}D_\mathcal{X}\} \cup \psi\theta\mathcal{P} \vdash \psi\theta K$. (2)
Now by *cut-elimination* over (1) and (2), $\psi\theta\{\forall\mathcal{X}.\,G_\mathcal{X} \Rightarrow D_\mathcal{X}\} \cup \psi\theta\mathcal{P} \vdash \psi\theta K$.

Incompleteness of "rule." The declarative reading of **rule** is *not*, however, equivalent to its operational definition, as the declarative version may succeed where the operational fails; that is,

$$\mathcal{P} \vdash \textbf{rule}\ G\ (\forall\mathcal{X}.\,G_\mathcal{X} \Rightarrow D_\mathcal{X})\ K$$

does *not* imply

$$\mathcal{P} \nvdash \textbf{rule}\ G\ (\forall\mathcal{X}.\,G_\mathcal{X} \Rightarrow D_\mathcal{X})\ K$$

even up to the usual deterministic limitations of logic programming (Chapter 3). This is because **rule**'s assumption $\forall\mathcal{X}'\,\forall\mathcal{Y}.\,\sigma_{\mathcal{X}\mathcal{Y}}\theta D_\mathcal{X}$ is typically less general than $\forall\mathcal{X}.\,\theta D_\mathcal{X} \Leftarrow \theta G_\mathcal{X}$, and thus K may follow from the latter, but not from the former. But this is, of course, the whole purpose of **rule**: to focus search by making use of a *selected consequence* $(\forall\mathcal{X}'\,\forall\mathcal{Y}.\,\sigma_{\mathcal{X}\mathcal{Y}}\theta D_\mathcal{X})$ of the general assumption $(\forall\mathcal{X}.\,\theta D_\mathcal{X} \Leftarrow \theta G_\mathcal{X})$, which, by itself, may be too powerful to be computationally useful.

5.3.4 Implementation Issues

λProlog constraints. In our discussion, we have thus far ignored a problem posed by λProlog's higher-order nature: higher-order variables, in addition to being instantiated, can accumulate constraints in the course of computation. These constraints are essential for higher-order unification, and have to be represented in the forms manipulated by **rule**. The actual form for **rule**'s assumption is $\forall\mathcal{Y}.\,\sigma_{\mathcal{X}\mathcal{Y}}\theta D_\mathcal{X} \Leftarrow \exists\mathcal{Z}.\,C_\mathcal{Z}$, where $C_\mathcal{Z}$ is the current set of constraints, and $\mathcal{Z}$ represents all variables occurring only in $C_\mathcal{Z}$. A similar solution for **assert** has been proposed for the more general constraint logic programming language CLP($\Re$)[12] in [61]: the constraints are therein reduced, to as great an extent as possible to the variables occurring in in the clause to be assumed, and then added as 'guards' to that clause. (See also §9.2.2.)

[12]CLP($\Re$) supports more general constraints such as those expressed within arithmetic inequalities [70].

```
rtp ⇐ rule (infer R)
           (∀R. infer R ⇒ clause R)
           (R = false; rtp).

infer R' ⇐ clause P, clause Q, resolve P Q R, simpl R R', (R' = false; keep? R').

resolve (P ; Q)   S        (P ; R) ⇐ resolve Q S R.
resolve (P ; Q)   S        (Q ; R) ⇐ resolve P S R.
resolve S         (P ; Q)  (P ; R) ⇐ resolve Q S R.
resolve S         (P ; Q)  (Q ; R) ⇐ resolve P S R.
resolve P         (not P)  false.
resolve (not P)   P        false.

keep? R ⇐ write R, write_string "Keep? : ", read λG. G.
```

Figure 5.1: Rudimentary Resolution Theorem Prover.

5.3.5 Example: "lemma"

Versions of our scoped **lemma** introduced in §5.2.2 may now be defined in terms of **rule** for both the non-committing case:

```
lemma E K ⇐ rule E (∀E. E ⇒ E) K.
```

and the committing:

```
lemma E K ⇐ rule E (∀E. E ⇒ (E ⇐ !)) K.
```

5.3.6 Example: Resolution

Consider **rtp**, a rudimentary resolution theorem prover, given in Figure 5.1. The predicate **clause** enumerates disjunctive expressions to be resolved, such as

```
clause (p x y; not (q y x)).
clause (q a z).
```

resolve blindly resolves its first two arguments, yielding a resolvent R, which is then simplified by **simpl** (whose clauses may be found in Appendix A.1). To illustrate,

```
? - resolve (p x y ; not (q y x)) (q a z) R, simpl R R'.
```

instantiates $R' = \mathbf{p}\ x\ \mathbf{a}$. To avoid infinitely re-deriving the same clause, the user is queried by the predicate **keep?** to determine whether R' should be used or discarded. **rtp** succeeds if it is able to derive a contradiction ($R' =$ **false**). **rtp** first invokes **infer**, which produces a resolvent of two clauses. If either $R' =$ **false** or **keep?** R' succeeds (*i.e.*, the user enters **true**), **infer** R succeeds. **rtp** then makes the forward step **infer** $R \Rightarrow$ **clause** R, and assumes the universal closure of **clause** R before recursively calling **rtp**. (A more complete implementation of **rtp** may be found in Appendix A.2.)

$$
\begin{array}{llll}
\textbf{typeof} & (\textbf{if}\ E\ F\ H) & A & \Leftarrow\ \textbf{typeof}\ E\ \textbf{bool},\ \textbf{typeof}\ F\ A,\ \textbf{typeof}\ H\ A. \\
\textbf{typeof} & (\textbf{lam}\ F) & (A \longrightarrow B) & \Leftarrow\ \forall x.\, \textbf{typeof}\ x\ A \Rightarrow \textbf{typeof}\ (Fx)\ B. \\
\textbf{typeof} & (\textbf{appl}\ F\ E) & B & \Leftarrow\ \textbf{typeof}\ F\ (A \longrightarrow B),\ \textbf{typeof}\ E\ A. \\
\textbf{typeof} & (\textbf{let}\ E\ F) & B & \Leftarrow\ \textbf{typeof}\ E\ A,\ \textbf{typeof}\ (FE)\ B. \\
\textbf{typeof} & (\textbf{fix}\ F) & A & \Leftarrow\ \forall x.\, \textbf{typeof}\ x\ A \Rightarrow \textbf{typeof}\ (Fx)\ A.
\end{array}
$$

Figure 5.2: ML Type Inference

5.3.7 Example: ML-style Type Inference

As a further application of **rule**, consider the example of programming ML-style type inference,[13] as implemented by Hannan & Miller [58]. Some of the more interesting rules for type inference are included within Figure 5.2. We are particularly interested in type inference over the ML **let** construct. We represent (**let** $x = E$ **in** Fx) within λProlog as **let** $E\ F$, where F is a λ-abstraction. (This reverses the order of arguments used within Hannan & Miller's representation.) Type inference for this construct can be captured by the following λProlog clause [58]:

$$\textbf{typeof}\ (\textbf{let}\ E\ F)\ B \Leftarrow \textbf{typeof}\ E\ A,\ \textbf{typeof}\ (FE)\ B.$$

The problem with the above formulation is that the type of E is computed once (to insure that it is indeed typable), and then thrown away. Instances of E are then re-typed at each occurrence of x within $\lambda x.Fx$. This is necessary because the type of E, namely A, could be polymorphic — *i.e.*, contain variables such as the $C \longrightarrow C$ typing of the identity $\lambda x.x$. Without this re-computation, a polymorphic E can only be assigned one typing (*e.g.*, **int** $\longrightarrow$ **int**), since in the course of matching that type, the logical variable C would be instantiated to **int**, thus preventing it from matching, say, **bool** $\longrightarrow$ **bool** later.

Now consider another formulation

$$
\begin{array}{l}
\textbf{typeof}\ (\textbf{let}\ E\ F)\ B \Leftarrow \forall x.\, \textbf{typeof}\ E\ A, \\
\qquad\qquad (\forall A.\, \textbf{typeof}\ x\ A \Leftarrow \textbf{typeof}\ E\ A) \Rightarrow \textbf{typeof}\ (F\,x)\ B.
\end{array}
$$

As in the previous encoding, the initial **typeof** $E\ A$ insures that E has a valid typing (which is necessary in the case that the argument x does not occur in the body F). Now, however, rather than type FE, we type Fx using the additional rule

$$\forall A.\, \textbf{typeof}\ x\ A \Leftarrow \textbf{typeof}\ E\ A.$$

[13] ML, a polymorphic programming language, is introduced within [89] and standardized within [90]. ML-style type inference is akin to type inference over the simply-typed λ-calculus of §3.2, except that ML includes the **let** construct (discussed below).

This version simply separates the re-computation of E's type from that of typing F. Just as before, different occurrences of x may be given different types, and, just as before, the type of x (and hence the type of E) is re-computed from scratch at every occurrence.

Once the re-computation has been separated, however, it can be avoided entirely using the universal generalization and limited amount of forward reasoning afforded by **rule**:

$$\begin{aligned}\textbf{typeof}\ (\textbf{let}\ E\ F)\ B \Leftarrow \forall x.\ \textbf{rule}\ &(\textbf{typeof}\ E\ A)\\ &(\forall A.\ \textbf{typeof}\ x\ A \Leftarrow \textbf{typeof}\ E\ A)\\ &(\textbf{typeof}\ (Fx)\ B).\end{aligned}$$

This makes an assumption of the form $\forall\mathcal{Y}.\textbf{typeof}\ x\ A$ while inferring the type of the body Fx. $\mathcal{Y}$ includes exactly those type variables in A which are not free in any assumption, thus directly expressing the restriction on the type inference rule for **let**. We do not lose any solutions, since ML has the principal type property, and therefore all solutions to **typeof** $E\ A'$ are instances of the assumption $\forall\mathcal{Y}.\textbf{typeof}\ E\ A$.

5.4 Explanation-Based Learning (EBL)

The **rule** construct has allowed us to write programs which could not be straightforwardly expressed in λProlog, such as the resolution theorem prover, as well as allowed us to formulate programs more efficiently, such as type inference for ML. Moreover, the ideas behind **rule** carry over to the problem of explanation-based generalization and learning, which is the topic of this section.

Assimilation bridges the gap between explanation-based *generalization* and explanation-based *learning*, where the latter additionally requires a means for incorporating generalizations within the logic program. The programmer controls EBG via extensions of **lemma** and **rule** — **lemma_ebg** and **rule_ebg**, which behave analogously except that their assumptions are instead explanation-based generalizations. And as before, **lemma_ebg** will turn out to be a special case of its more general counterpart, **rule_ebg**.

The following illustrates how the explanation-based generalizations of Chapter 4 could be derived and then assumed in the scope of some further computation K: for the suicide example of §4.2, the solution is

$$?-\ \textbf{lemma\_ebg}\ (\textbf{kill john john})\ K.$$

and for the symbolic integration problem of §4.4,

$$?-\ \textbf{lemma\_ebg}\ (\textbf{intgr}\ (\lambda x.3 * x^2 + \cos x)\ h)\ K.$$

5.4.1 The "rule_ebg" Construct

Rather than first considering **lemma_ebg**, we move directly to the general case, **rule_ebg**. The general form for **rule_ebg** is

$$\textbf{rule\_ebg}\ G\ (\Box\,\forall\mathcal{X}.\,G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ K$$

The formulation of **rule_ebg** depends upon the $\Box$ operator introduced in §4.3: recall that the necessary truth of rules derived through EBG is captured with the $\Box$ prefix. To insure the modal validity of **rule_ebg**'s assumption, we require that the forward inference step also be necessarily true.[14]

Operational reading. **rule_ebg**'s operational interpretation is as follows:

1. Solve $\mathcal{P} \Vdash_\theta G$ with EBG enabled. If this fails, backtrack. Otherwise, it yields some explanation-based generalization

   $$\Box\,\forall\mathcal{Y}.\,\theta GG_{\mathcal{Y}} \Leftarrow \theta DD_{\mathcal{Y}}.$$

   where $GG_{\mathcal{Y}}$ is the generalized query, $DD_{\mathcal{Y}}$ captures the preconditions of the generalization (the choice of the symbol 'DD' will be motivated within §8.6), $\mathcal{Y}$ may appear free within $GG_{\mathcal{Y}}$ and $DD_{\mathcal{Y}}$, and $\theta GG_{\mathcal{Y}}$ necessarily has θG as an instance. The latter is a consequence of the EBG algorithm itself: the original query must be an instance of the generalized query (§8.6).

2. Find minimal substitutions $\sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{Y}}$ such that $\mathbf{dom}(\sigma_{\mathcal{X}}) \subseteq (\mathcal{X})$, $\mathbf{dom}(\sigma_{\mathcal{Y}}) \subseteq (\mathcal{Y})$, and $\sigma_{\mathcal{X}}\theta G_{\mathcal{X}} = \sigma_{\mathcal{Y}}\theta GG_{\mathcal{Y}}$.

3. Let $\mathcal{X}' = \mathcal{X} - \mathbf{dom}(\sigma_{\mathcal{X}})$.

4. Let $\mathcal{Y}' = \mathcal{Y} - \mathbf{dom}(\sigma_{\mathcal{Y}})$.

5. Solve

   $$\{\Box\,\forall\mathcal{X}'.\,\forall\mathcal{Y}'.\,\sigma_{\mathcal{X}}\sigma_{\mathcal{Y}}\theta(D_{\mathcal{X}} \Leftarrow DD_{\mathcal{Y}})\} \cup \theta\mathcal{P} \vdash_\psi \theta K.$$

(As with **rule**, we may again rely upon λProlog unification to derive a more logic programming-oriented interpretation of **rule_ebg**

1. Solve G with EBG enabled, resulting in the explanation-based generalization

   $$\Box\,\forall\mathcal{Y}.\,\theta GG_{\mathcal{Y}} \Leftarrow \theta DD_{\mathcal{Y}}.$$

[14] We do not use !! in place of $\Box\,\forall$ for **rule_ebg**'s forward inference step as the former is only permitted at the top-level of program clauses.

2. Create new logical variables $\hat{\mathcal{X}}$ for each universal variable in $\mathcal{X}$, and then substitute $\hat{\mathcal{X}}$ for $\mathcal{X}$ in $G_{\mathcal{X}}$ and $D_{\mathcal{X}}$, yielding $G_{\hat{\mathcal{X}}}$ and $D_{\hat{\mathcal{X}}}$.

3. Create new logical variables $\hat{\mathcal{Y}}$ for each universal variable in $\mathcal{Y}$, and then substitute $\hat{\mathcal{Y}}$ for $\mathcal{Y}$ in $GG_{\mathcal{Y}}$ and $DD_{\mathcal{Y}}$, yielding $GG_{\hat{\mathcal{Y}}}$ and $DD_{\hat{\mathcal{Y}}}$.

4. Unify $GG_{\hat{\mathcal{Y}}}$ and $G_{\hat{\mathcal{X}}}$.

5. Solve $(\Box\, \forall\hat{\mathcal{Y}}.\, \forall\hat{\mathcal{X}}.\, D_{\hat{\mathcal{X}}} \Leftarrow DD_{\hat{\mathcal{Y}}}) \Rightarrow K.)$

Declarative reading. While **rule_ebg** differs from **rule** in the obvious way, the same declarative reading is applicable to both! This is because in the same sense as **rule**, **rule_ebg** does not permit assumptions not derivable from the logic program. The proof relies upon the validity of our EBG algorithm (established in §8.6), and then employs techniques analogous to those used in the proof for **rule** (§5.3.3). We omit this proof as it requires deriving formal soundness properties under an inference system extended with $\Box$ (addressed in §8.3) and with EBG (not addressed), but we do provide a formal characterization of $\lambda^{\Box}$Prolog generalization in the form of an abstract interpreter in §8.6.

As is the case with **rule**, **rule_ebg** must also take λProlog constraints into account. The situation is handled analogously. (See also §9.2.2.)

5.4.2 Example: "lemma_ebg"

We may now define **lemma_ebg** in terms of **rule_ebg**:

$$\textbf{lemma\_ebg}\ E\ K \Leftarrow \textbf{rule\_ebg}\ E\ (\Box\, \forall E.\, E \Rightarrow E)\ K.$$

Similarly, a committing version may be defined as

$$\textbf{lemma\_ebg}\ E\ K \Leftarrow \textbf{rule\_ebg}\ E\ (\Box\, \forall E.\, E \Rightarrow (E \Leftarrow !))\ K.$$

Additional illustrations of **rule_ebg** appear in the remaining chapters.

Chapter 6

Search Control via Tactics and Programmable Learning

The integration example we presented within §4.4 relied upon logic programming's implicit search to solve queries. Additional levels of search control need not, however, interfere with the underlying process of EBG! We demonstrate this by implementing a *tactic-based* solution of the symbolic integration problem. Search is controlled within a tactic-based theorem prover (or problem solver) by requiring the user to *a priori* or interactively specify a combination of proof steps, or *tactics*, with which to attempt the derivation of a goal [50, 20]. This combination of tactics guides the construction of an actual proof (or problem solution).

6.1 Example: Tactic-Style Symbolic Integration

Once again, our presentation herein focuses upon the most relevant and interesting aspects of the example; the unabridged tactic-based problem solver may be found in Appendix A.5.

Tactics are simply named rules: for the integration domain, we have

```
!! tac constant      (intgr (λx.a)        (λx.a * x))
                     true.
!! tac power         (intgr (λx.x^a)      (λx.x^(a+1)/(a + 1)))
                     true.
!! tac constant_left (intgr (λx.a * fx)   (λx.a * f'x))
                     (intgr f             f').
!! tac plus          (intgr (λx.fx + hx)  (λx.f'x + h'x))
                     (intgr f             f',
                      intgr h             h').
   tac cos_tac       (intgr cos           sin)
                     true.
```

Tactics perform goal reduction: the input goal G_{in} (2nd argument) is reduced to a more easily solved subgoal G_{out} (3rd argument).

To represent compositions of tactics, we have problem independent *meta*-tactics, or *tacticals*, such as

$$
\begin{array}{llllll}
!! & \textbf{tac} & \textbf{idtac} & G_{\text{in}} & G_{\text{in}}. & \\
!! & \textbf{tac} & (\textbf{then}\ T_1\ T_2) & G_{\text{in}} & G_{\text{out}} & \Leftarrow\ \textbf{tac}\ T_1\ G_{\text{in}}\ G_{\text{med}},\ \textbf{tac}\ T_2\ G_{\text{med}}\ G_{\text{out}}. \\
!! & \textbf{tac} & (\textbf{orelse}\ T_1\ T_2) & G_{\text{in}} & G_{\text{out}} & \Leftarrow\ \textbf{tac}\ T_1\ G_{\text{in}}\ G_{\text{out}};\ \textbf{tac}\ T_2\ G_{\text{in}}\ G_{\text{out}}. \\
!! & \textbf{tac} & (\textbf{try}\ T) & G_{\text{in}} & G_{\text{out}} & \Leftarrow\ \textbf{tac}\ (\textbf{orelse}\ T\ \textbf{idtac})\ G_{\text{in}}\ G_{\text{out}}. \\
!! & \textbf{tac} & (\textbf{complete}\ T) & G_{\text{in}} & \textbf{true} & \Leftarrow\ \textbf{tac}\ T\ G_{\text{in}}\ \textbf{true}. \\
!! & \textbf{tac} & (\textbf{repeat}\ T) & G_{\text{in}} & G_{\text{out}} & \Leftarrow\ \textbf{tac}\ (\textbf{orelse}\ (\textbf{then}\ T\ (\textbf{repeat}\ T))\ \textbf{idtac})\ G_{\text{in}}\ G_{\text{out}}.
\end{array}
$$

Tacticals are applied to compound goals (*i.e.*, those containing logical connectives) via **maptac**:

$$
\begin{array}{llllll}
!! & \textbf{tac} & (\textbf{maptac}\ T) & \textbf{true} & \textbf{true}. & \\
!! & \textbf{tac} & (\textbf{maptac}\ T) & (G_{\text{in1}}\ , G_{\text{in2}}) & G_{\text{out}} & \Leftarrow\ !,\ \textbf{tac}\ (\textbf{maptac}\ T)\ G_{\text{in1}}\ G_{\text{out1}}, \\
 & & & & & \qquad \textbf{tac}\ (\textbf{maptac}\ T)\ G_{\text{in2}}\ G_{\text{out2}}, \\
 & & & & & \qquad \textbf{simpl}\ (G_{\text{out1}}\ , G_{\text{out2}})\ G_{\text{out}}. \\
!! & \textbf{tac} & (\textbf{maptac}\ T) & (G_{\text{in1}}\ ; G_{\text{in2}}) & G_{\text{out}} & \Leftarrow\ !,\ \textbf{tac}\ (\textbf{maptac}\ T)\ G_{\text{in1}}\ G_{\text{out1}}, \\
 & & & & & \qquad \textbf{tac}\ (\textbf{maptac}\ T)\ G_{\text{in2}}\ G_{\text{out2}}, \\
 & & & & & \qquad \textbf{simpl}\ (G_{\text{out1}}\ ; G_{\text{out2}})\ G_{\text{out}}. \\
!! & \textbf{tac} & (\textbf{maptac}\ T) & G_{\text{in}} & G_{\text{out}} & \Leftarrow\ \textbf{tac}\ T\ G_{\text{in}}\ G_{\text{med}}, \\
 & & & & & \qquad \textbf{simpl}\ G_{\text{med}}\ G_{\text{out}}.
\end{array}
$$

where the clauses for **simpl** may be found in Appendix A.1. (The above tacticals were to a large degree borrowed from Felty [44, pp.143–149].)

We augment the above with a special interactive tactical:

$$
\begin{array}{llllll}
!! & \textbf{tac} & \textbf{interactive} & G_{\text{in}} & G_{\text{out}} & \Leftarrow\ \textbf{write\_string}\ \text{“Goal to be reduced : ”},\ \textbf{write}\ G_{\text{in}}, \\
 & & & & & \qquad \textbf{newline},\ \textbf{write\_string}\ \text{“Enter tactic/tactical : ”}, \\
 & & & & & \qquad \textbf{read}\ \lambda T.\ \textbf{tac}\ T\ G_{\text{in}}\ G_{\text{med}},\ ((G_{\text{med}} = \textbf{true}, G_{\text{out}} = \textbf{true}) \\
 & & & & & \qquad\qquad ;\ \textbf{tac interactive}\ G_{\text{med}}\ G_{\text{out}}).
\end{array}
$$

Now to solve the query

$$
?-\ \textbf{tac}\ \ \textbf{interactive}\ \ (\textbf{intgr}\ \ (\lambda x.2 * (3 * \cos x))\ \ h)\ \ G_{\text{out}}.
$$

we could enter the series of tactics **constant_left**, **constant_left**, and **cos_tac** as prompted; or equally, the tactical

$$
\textbf{then}\ (\textbf{repeat}\ \textbf{constant\_left})\ \textbf{cos\_tac}
$$

yielding

$$
\begin{array}{lll}
H & = & \lambda x.2 * (3 * \sin x) \\
G_{\text{out}} & = & \textbf{true}
\end{array}
$$

Entering the same series of tactics or the same tactical to the query

?– **lemma_ebg** (**tac interactive** (**intgr** $(\lambda x.2*(3*\cos x))$ h) G_{out})
top.

leads to the assimilation of the explanation-based generalization

!! **tac interactive** (**intgr** $(\lambda x.a*(b*fx))$ $(\lambda x.a*(b*f'x))$
(**intgr** f f').

(In §6.3 we discuss why it is *not* desirable for **interactive** to appear within the generalization.)

As a somewhat more complex illustration, the query

?– **lemma_ebg** (**tac interactive** (**intgr** $(\lambda x.2+(3*x^2))$ h) G_{out})
top.

when solved, for example, by the tactical

then plus (maptac (orelse constant (then constant_left power)))

assimilates the generalization

!! **tac interactive** (**intgr** $(\lambda x.a+(b*x^c))$ $(\lambda x.(a*x)+(b*x^{c+1}/(c+1))))$
true.

6.2 Level of Generalization

As one would expect, the above explanation-based generalizations are applicable to problems not addressed by the original tactical:

$\lambda x.2*(3*\sin x)$
$\lambda x.a*(y*(3*x))$

This is, of course, because tactics of the training theory are abstracted (as well as constants of the original goal).

At the same time, the derived rules do not cover the range of problems for which the given *tacticals* are applicable: consider that the first tactical

then (repeat constant_left) cos_tac

solves each of the integrals

$$\lambda x.\cos x$$
$$\lambda x.4 * \cos x$$
$$\lambda x.2 * (3 * \cos x)$$
$$\lambda x.a * (y * (3 * \cos x))$$

and so on.

Because the tactical- or meta-level is formulated completely within $\mathcal{D}$, generalization does not occur at that level; instead generalization is confined to the tactic- or rule-level. This is, of course, exactly what we were after when we set out to make the additional level of search control transparent from the perspective of EBG. Alternative formulations could produce generalizations at the tactical-level, but those derived rules are more likely to be so general that they would be difficult to apply. [37].

6.3 Level of Assimilation

As discussed within Chapter 5, under the traditional approach to learning, the problem solver produces and assimilates generalizations in the course of solving queries. Such an approach to assimilation is, however, problematic for tactic-based paradigms. In the above example, although generalization occurs only at the level of *tactics*, the derived rule nevertheless contains a reference to the *tactical* **interactive**. If we are to maintain a strict separation of the rule-level and meta-level, it does not make sense to assimilate a generalization encompassing both levels. Rather, a slightly modified generalization could be assimilated at the rule-level as a *derived tactic:*

$$!!\ \textbf{tac}\ \ \textbf{constant\_left\_two}\ \ (\textbf{intgr}\ \ (\lambda x.a * (b * fx))\ \ (\lambda x.a * (b * f'x)))$$
$$(\textbf{intgr}\ \ f\ \ f').$$

Moreover, this assimilation of a derived tactic can be achieved through the limited forward reasoning provided by **rule_ebg**:

$$?-\ \ \textbf{rule\_ebg}\ \ (\textbf{tac interactive}\ (\textbf{intgr}\ (\lambda x.2 * (3 * \cos x))\ h)\ G_{\text{out}})$$
$$\forall G_{\text{in}}\ \forall G_{\text{out}}.\ \textbf{tac interactive}\ G_{\text{in}}\ G_{\text{out}} \Rightarrow \textbf{tac constant\_left\_two}\ G_{\text{in}}\ G_{\text{out}}$$
$$\textbf{top}.$$

The point here, and it is an important one, is that it is the user (or client program), rather than the problem solver, which is in a position to control assimilation in this situation. If we were to instead directly assimilate the original generalization, we compromise the predicate **interactive** in that a subsequent invocation might no longer prompt the user; that is, we compromise the user's control over search.

This example reinforces our belief that for such applications EBG should be a *feature* of the language in which problem solvers are coded, rather than a 'black box' within the problem solving architecture. In other words, what is required is a language in which one can *program* the learning mechanism. By providing the programmer with an explicit means

to control generalization and assimilation, we defer the difficult problem of determining when to generalize and assimilate [72]. Client programs have the potential advantage of bringing domain knowledge and user interaction to bear in determining what is to be learned. This concept of programming generalization and learning within the same language in which problem solving and interaction occur is markedly different from what we label 'black-box' learning. Hence our approach stands in contrast to systems such as Soar [75], Prodigy [92], and LEAP [96] in which learning is largely relegated to the system.

6.4 Operationality *vs.* □ – Revisited

While §4.5 illustrated that both □ and operationality criteria serve to define EBG's generalized proofs (and hence its results), the tactic example above demonstrates that the mechanisms are *not* interchangeable: consider that a formulation of the integration domain that replaces □ with operationality criteria (defined via the predicate **oper**) requires specifying

oper (**tac** **interactive** (**intgr** cos sin) **true**).

The problem is again that this definition forces the mixing of the rule- and meta-level, thereby violating the modularity of our encoding.

6.5 An EBG Tactical

As presented within §6.3, the following query represents a way to perform EBG over interactive tactic-based problem solving:

?– **rule_ebg** (**tac** **interactive** (**intgr** $(\lambda x.2*(3*\cos x))$ h) G_{out})
($\forall G_{\text{in}}\, \forall G_{\text{out}}$. **tac** **interactive** G_{in} G_{out} ⇒ **tac** **constant_left_two** G_{in} G_{out})
top.

EBG need not, however, be separated from the meta-level: consider the special generalization tactical

!! **tac** (**ebg_tac** *Tac*) G_{in} G_{out} ⇐ **rule_ebg** (**tac** **interactive** G_{in} G_{med})
($\forall G_{\text{in}}\, \forall G_{\text{med}}$. **tac** **interactive** G_{in} G_{med}
⇒ **tac** *Tac* G_{in} G_{med})
(**tac** **interactive** G_{med} G_{out})

At any point in the interactive solution of a goal G_{in}, the user may initiate EBG via **ebg_tac**, which takes the name *Tac* of the tactic to be derived as an argument (*e.g.*, **constant_left_two**). **rule_ebg**, in turn, recursively invokes **interactive** to reduce the initial goal G_{in} to some other goal G_{med}. When this nested invocation returns (or 'pops'), which would result from the user entering **idtac** as prompted, **rule_ebg** uses the resulting explanation-based generalization to derive a new tactic *Tac*. As a result of the forward chaining step, this newly derived tactic is assimilated, and thus made available (at the user's

request) for application within the solution of G_{med}. Moreover, if we want to retain the result of EBG after solving G_{med}, simply replace the third argument of **rule_ebg** with

tac interactive G_{med} G_{out}, **top**

(A further illustration of the EBG tactical may be found in Appendix A.6.)

Chapter 7

Program Transformation and Apprentice Learning

As introduced within §2.1, one paradigm for formal program development is that of *program transformation* [13, 68, 42, 102]. Under a transformational approach, an abstract specification of an algorithm is refined, or *specialized*, through a sequence of formal elaboration steps, or *transformations*, into a program with acceptable performance. The resulting sequence of transformations, or *meta-program*, along with the initial specification serve as a *derivation*, or justification, of the optimized program.

In that they encode named incremental problem solving steps subject to composition, program transformations are akin to tactics. The difference is, of course, that transformations operate on programs (or subexpressions of programs) rather than upon goals. Also, tactics embody theorem proving steps, which are generally directional (reducing goals to more easily solved subgoals), while there is typically no clear directionality to transformations. Typically, transformations map one program to a functionally equivalent version that may have different performance characteristics.

7.1 Example: Tail Recursion

We illustrate EBG over a transformation system we have applied to induce tail recursion in certain situations.[1] (From a tail recursive version, an iterative form could easily be derived.) We begin with a functional specification of the factorial program:

$$\begin{array}{l}\textbf{fix}\ \lambda \mathit{fact}.\ \textbf{lam}\ \lambda n.\ \textbf{if}\ (\textbf{equals}\ n\ 0) \\ \qquad\qquad\qquad\qquad 1 \\ \qquad\qquad\qquad\qquad (\textbf{appl}\ \mathit{fact}\ \ (n-1)) * n\end{array}$$

[1] This example is treated more abstractly within Dietzen & Scherlis [33], among others.

The above is a λProlog abstract syntax for a simple functional language. The constructs **lam** and **appl** represent object-level λ-abstraction and application, respectively. (The incorporation of explicit notation allows us to distinguish meta- and object-level. This provides programmer control over operations such as β-reduction: that is, we can write (**appl** (**lam** $\lambda x.\, x$) 1) without λProlog performing the reduction, as it would for the direct representation $((\lambda x.\, x)1) \Rightarrow_\beta 1$. β-reduction is then handled explicitly by replacing (**appl** (**lam** f) x) with fx.) Finally, the fixpoint or recursion operator **fix** is 'applied' by substituting its body for each occurrence of the bound identifier within its body.

The derivation proceeds by applying transformations to this specification. For example, the following transformation replaces an occurrence of e with $op\ e\ \mathbf{z}$, where $\mathbf{z}$ is a right identity of op (for example, mapping $\mathbf{a}$ to $\mathbf{a} + 0$):

$$!!\ \textbf{add\_id\_right}\ \ op\ \ C\ \ (C\,e)\ \ (C\,(op\,e\,z))\ \ \Leftarrow\ \ \textbf{right\_identity}\ op\ z.$$

The third and forth arguments match the input and output object programs, respectively. The second argument C specifies a *context* — *i.e.*, the particular subexpression of the input program to be transformed. These higher-order context variables serve to formally encode subterm or *occurrence* selection, which might, for example, result from "pointing with a mouse" [106]. This represents yet another application of higher-order representation language: the formal expression of occurrences. For example, within the following invocation of the transformation

$$?-\ \ \textbf{add\_id\_right}\ \ (\lambda x.\lambda y.\, x + y)\ \ (\lambda g.g * h)\ \ (\mathbf{a} * \mathbf{b})\ \ F_{\text{out}}.$$

the context variable is $\mathrm{C} = \lambda g.\, g * h$. From the definition of **add_id_right** above, C is applied to e and then matched against the input $\mathbf{a} * \mathbf{b}$; that is,

$$\begin{array}{rcl} C\,e & = & (\lambda g.g * h)\,e \\ & =_\beta & e * h \\ & = & \mathbf{a} * \mathbf{b} \end{array}$$

Thus, e is instantiated to $\mathbf{a}$ and h to $\mathbf{b}$. Now, given that

$$\textbf{right\_identity}\ (\lambda x.\lambda y.\, x + y)\ 0$$

the output F_{out} is instantiated as follows:

$$\begin{array}{rcl} F_{\text{out}} & = & C\,(op\ e\ z) \\ & = & (\lambda g.\, g * \mathbf{b})\,((\lambda x.\lambda y.\, x + y)\ \mathbf{a}\ 0) \\ & =_\beta & (\lambda g.\, g * \mathbf{b})\,(\mathbf{a} + 0) \\ & =_\beta & (\mathbf{a} + 0) * \mathbf{b} \end{array}$$

The full derivation, which consists of a sequence of ten such transformation rules and the associated contexts, constitutes a meta-program — *i.e.*, a program that manipulates an object program such as *fact*. Like tacticals, meta-programs may be specified *a priori*, or

constructed interactively. Ideally, interactive construction of meta-programs consists of alternatively naming a transformation rule, and then selecting an appropriate context. And ideally, context selection would be derived by translating mouse input into the appropriate higher-order context. However, as eLP currently lacks the necessary interface to interpret mouse events, contexts have herein been hand-coded. We first describe an *a priori* meta-program **tail_rec** embodying this derivation. We later provide, in Appendix A.6, an implementation supporting the interactive construction of a meta-program equivalent to **tail_rec**.

7.1.1 Derivation

We now enumerate the individual steps of the tail-recursive *fact* derivation. Our discussion will focus upon the abstract nature of the transformations, rather than upon the low-level details of transformation application itself; the latter was treated to some extent for **add_id_right** above. After grasping this abstract description of the derivation, the reader may then want to review the $\lambda^{\square}$Prolog representation of these transformations (Figure 7.1), and their application with the appropriate contexts via the meta-program **tail_rec** (Figure 7.2). However, for many readers the intimate details of *both* the abstract derivation and its $\lambda^{\square}$Prolog counterpart may prove too tedious to be of interest. Indeed, there is nothing new in these transformations, except, to some degree, their representation within higher-order language. (The case for using higher-order language to represent program transformations is argued by Huet & Lang [68], and more recently by Pfenning & Elliott [106], and Hannan & Miller [59].) It is the application of higher-order EBG to the whole process which is our contribution. For those readers more interested in the latter, I suggest you skip ahead to the discussion of §7.1.2.

0. We begin with the initial definition of *fact*.

$$\textbf{fix}\ \lambda \mathit{fact}.\ \textbf{lam}\ \lambda n.\ \textbf{if}\ (\textbf{equals}\ n\ 0)\ 1\ (\textbf{appl}\ \mathit{fact}\ (n-1)) * n$$

1. η-expand term in the object-language; that is, insert a **lam** and an **appl**. ('...' elides the body of *fact*.)

$$\textbf{lam}\ \lambda n.\ \textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}.\ \ldots)\ n$$

   (The above n is distinct from that within *fact*'s body.)

2. Insert a multiplication by 1. This transformation relies upon **right_identity** $(\lambda x.\lambda y.\ x * y)\ 1$.

```
lam λn. (appl (fix λfact. ...)
              n) * 1
```

3. Abstract over 1; that is, make it a parameter. This introduces a second argument which is to become the accumulator within the eventual tail recursive version.

```
appl (lam λm. lam λn.
        (appl (fix λfact. ...)
              n) * m)
     1
```

4. Name the resulting two argument function $fact_1$: since **fix** specifies the expansion of recursive functions, one may think of it as a mechanism for function definition. This initial definition of $fact_1$ will be used later in the derivation.

```
appl (fix λfact1. lam λm. lam λn.
        (appl (fix λfact. ...)
              n) * m)
     1
```

5. *Unfold* the recursive definition of *fact*; that is, expand the fixpoint operator once.

```
appl (fix λfact1. lam λm. lam λn.
        (appl (lam λn'. if (equals n' 0)
                           1
                           ((appl (fix λfact. ...) n' - 1) * n'))
              n) * m)
     1
```

6. β-reduction in the object-language; that is, (**appl** (**lam** $\lambda n'.\ fn'$) n) $\Rightarrow_\beta$ fn.

```
appl (fix λfact1. lam λm. lam λn.
        (if (equals n 0)
            1
            ((appl (fix λfact. ...) n - 1) * n))
         * m)
     1
```

7. Distribute * over **if**.

```
appl (fix λfact1. lam λm. lam λn.
        if (equals n 0)
           1 * m
           ((appl (fix λfact. ...) n - 1) * n) * m)
     1
```

8. Simplify the *then*-clause using the fact that **left_identity** $(\lambda x.\lambda y.\, x * y)\ 1$.

$$
\begin{array}{l}
\textbf{appl}\ (\textbf{fix}\ \lambda fact_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n. \\
\qquad\quad \textbf{if}\ (\textbf{equals}\ n\ 0) \\
\qquad\qquad m \\
\qquad\qquad ((\textbf{appl}\ \ (\textbf{fix}\ \lambda fact.\ ...)\ \ n-1) * n) * m) \\
\qquad 1
\end{array}
$$

9. Re-associate the multiplicative expression of the *else*-clause, since **associative** $\lambda x.\lambda y.\, x * y$.

$$
\begin{array}{l}
\textbf{appl}\ (\textbf{fix}\ \lambda fact_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n. \\
\qquad\quad \textbf{if}\ (\textbf{equals}\ n\ 0) \\
\qquad\qquad m \\
\qquad\qquad (\textbf{appl}\ \ (\textbf{fix}\ \lambda fact.\ ...)\ \ n-1) * (n * m)) \\
\qquad 1
\end{array}
$$

10. Observe that within step 9 the subexpression

$$
(\textbf{appl}\ \ (\textbf{fix}\ \lambda fact.\ ...)\ \ n-1) * (n * m))
$$

is a higher-order instance of the original definition of $fact_1$ given in step 4:

$$
\begin{array}{l}
\textbf{fix}\ \lambda fact_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n. \\
\quad (\textbf{appl}\ (\textbf{fix}\ \lambda fact.\ ...) \\
\qquad\quad n) * m)
\end{array}
$$

The only difference is the values of the arguments m and n. This means that we may *fold* the above expression into a $fact_1$ invocation.

$$
\begin{array}{ll}
\textbf{appl}\ (\textbf{fix}\ \lambda fact_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n.\ & \textbf{if}\ \textbf{equals}\ n\ 0 \\
 & m \\
 & (\textbf{appl}\ \ (\textbf{appl}\ \ fact_1\ \ (n * m))\ \ (n-1))) \\
\qquad 1 &
\end{array}
$$

This completes the derivation.

As mentioned above, each of the preceding transformation steps is formally represented in Figure 7.1. While we do not attempt a proof, we claim that these transformations are *correctness preserving* — *i.e.*, they do not change the functionality of the program.

7.1.2 Generalizing the Derivation

The **tail_rec** meta-program may be applied to *fact* through the query

$$
\begin{array}{l}
?- \quad \textbf{tail\_rec}\ (\lambda x.\lambda y.\ x * y) \\
\qquad\qquad (\textbf{fix}\ \lambda \mathit{fact}.\ \textbf{lam}\ \lambda n.\ \textbf{if}\ (\textbf{equals}\ n\ 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{appl}\ \mathit{fact}\ (n-1)) * n) \\
\qquad\qquad ?\ F_{\text{out}}.
\end{array}
$$

which yields the tail recursive expression

$$
\begin{array}{lll}
F_{\text{out}} & = & \textbf{appl}\ (\textbf{fix}\ \lambda \mathit{fact}_1.\ \textbf{lam}\ \lambda m.\ \textbf{lam}\ \lambda n.\ \textbf{if}\ (\textbf{equals}\ n\ 0) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad m \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{appl}\ (\textbf{appl}\ \mathit{fact}_1\ (n * m))\ (n-1))) \\
 & & \qquad 1
\end{array}
$$

For explanation-based generalization, we instead make the query

$$
\begin{array}{l}
?- \quad \textbf{lemma\_ebg}\ (\textbf{tail\_rec}\ (\lambda x.\lambda y.\ x * y) \\
\qquad\qquad\qquad\qquad (\textbf{fix}\ \lambda \mathit{fact}.\ \textbf{lam}\ \lambda n.\ \textbf{if}\ (\textbf{equals}\ n\ 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{appl}\ \mathit{fact}\ (n-1)) * n) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{top}.
\end{array}
$$

which leads to the assimilation of the following generalization:

$$
\begin{array}{l}
!!\ \textbf{tail\_rec}\ \mathit{op} \\
\qquad (\textbf{fix}\ \lambda f.\ \textbf{lam}\ \lambda y.\ \textbf{if}\ (H_1\ y) \\
\qquad\qquad\qquad\qquad\qquad\qquad a \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{op}\ (\textbf{appl}\ f\ (H_2\ y))\ (H_3\ y))) \\
\qquad (\textbf{appl}\ (\textbf{fix}\ \lambda f'.\ \textbf{lam}\ \lambda x.\ \textbf{lam}\ \lambda y.\ \textbf{if}\ (H_1\ y) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{appl}\ (\textbf{appl}\ f'\ (\mathit{op}\ (H_3\ y)\ x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (H_2\ y))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad b) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\
\qquad \Leftarrow\ \textbf{right\_identity}\ \mathit{op}\ b, \textbf{left\_identity}\ \mathit{op}\ a, \textbf{associative}\ \mathit{op}.
\end{array}
$$

The result produced by our prototype is not so elegantly expressed: it consists instead of a series of constraint equations. We took the liberty of collapsing them into their 'most obvious' solution above for presentation. The problem of more elegantly displaying these constraints requires further consideration; see §9.2.2.

In either form, however, the generalization may be applied to analogous programs such as list reversal:

$$
\begin{array}{l}
?- \quad \textbf{tail\_rec append} \\
\qquad\qquad (\textbf{fix}\ \lambda rev.\ \textbf{lam}\ \lambda l.\ \textbf{if}\ (\textbf{null}\ l) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{nil} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{append}\ (\textbf{appl}\ \ rev\ \ (\textbf{tl}\ l))\ ((\textbf{hd}\ l) :: \textbf{nil}))) \\
\qquad\qquad F_{\text{out}}.
\end{array}
$$

which instantiates[2]

$$
\begin{array}{l}
a = \textbf{nil} \\
b = \textbf{nil} \\
H_1 = \textbf{null} \\
H_2 = \textbf{tl} \\
H_3 = \lambda l.\ \textbf{hd}\ l :: \textbf{nil}
\end{array}
$$

yielding the tail-recursive version

$$
\begin{array}{l}
F_{\text{out}} = \textbf{appl}\ (\textbf{fix}\ \lambda rev_1.\ \textbf{lam}\ \lambda k.\ \textbf{lam}\ \lambda l.\ \textbf{if}\ (\textbf{null}\ l) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{appl}\ (\textbf{appl}\ \ rev_1\ \ (\textbf{append}\ ((\textbf{hd}\ l) :: \textbf{nil})\ k)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\textbf{tl}\ l)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{nil}
\end{array}
$$

The above result requires only the addition of a final simplification to make the reduction from (**append** ((**hd** l) :: **nil**) k) to ((**hd** l) :: k). Hence, the generalized *fact* derivation is sufficient for *rev* as well (except for the final simplification).

7.2 Expressiveness of Higher-order Generalization

The elegance of the preceding generalization is largely due to the expressiveness of our higher-order language. In particular, essential restrictions on the input program are implicit in the higher-order notation: (1) that the function argument y may not appear in the 'then' part of the **if**-statement, (2) that the function f may not be recursively invoked in the 'conditional' or 'then' parts of the **if**, and (3) that the recursive call to f within the 'else' branch must be the argument to a particular function *op* having special properties. These restrictions are not explicit in any single transformation step, but rather are spread over the sequence of transformations embodied by the generalization. Realizing a similar result within a first-order system would be complicated by the need for these checks.

Admittedly, even with the expressivity of higher-order language, program development by transformation is a very tedious business. But this is precisely why this domain represents

[2]While the derivation never establishes that $a \equiv b$, this follows from the fact that $a \equiv (op\ a\ b) \equiv b$ using **right_identity** *op* b and **left_identity** *op* a.

an attractive application for explanation-based generalization: by employing EBG to abstract derivations, one hopes to derive 'larger-grain' transformations — 'macro operators', if you will. Thus, it is our belief that EBG is one means by which to develop higher-level transformations, of which the preceding example is an illustration.

7.3 Apprentice Learning

The search space for the above derivation is so complex that without user guidance (*e.g.*, via an explicit meta-program, specified *a priori* or interactively), it would not be feasible for a system to 'discover' the sequence of transformations and their associated contexts with which to induce tail recursion. The transformation problem space is further complicated by the fact that it is the user who decides when a derived program is acceptably 'efficient' (in this case, when it is tail-recursive). Within the transformation paradigm, we are not in the situation of theorem proving where there are only two answers — "yes, a goal is provable" or "no, it is not." Instead, the role of the user is two-fold: to guide the derivation and to make value judgments upon the resulting programs. Currently we are so far from automating the latter that transformation systems will continue to depend upon user assistance.

That these value judgments are *not* represented within the transformations means they are *not* manifest in the resulting generalizations. There is an important underlying assumption here: namely that, a sequence of transformations which leads to a 'good' program in one particular case (*e.g.*, *fact*) is presumed to do the same for other programs to which it is applicable (*e.g.*, *rev*). However, as this 'goodness' exists outside of the transformations themselves, there is no guarantee that a derived rule indeed yields a 'good' program.[3]

Explanation-based generalization is often labeled 'speed-up' learning in that EBG extends the domain theory by constructing new rules in the deductive closure of that domain theory. In other words, under EBG nothing new may be proven, but the solution of problems covered by derived rules is (hopefully) quicker. With the incorporation of user interaction to address the problem of intractable search, this characterization of EBG becomes invalid: the resulting generalizations, while in the deductive closure of the rule set, are generally *not* accessible without user guidance. Here EBG serves as a vehicle to *transfer* knowledge from the user to the learner. The combination of learner and user, when viewed as a whole, still only accomplish speed-up learning. But, after a joint derivation of *fact*, the learner could handle *rev* without user assistance (presuming that the system could find the final simplification). That is, from the individual perspectives of the learner and user, more than speed-up learning has taken place [25, pp.151–153] [29, pp.304–305].

7.4 Other work

The above is reminiscent of the *learning apprentice system* defined by Mitchell [94] which *LEAP*, a learning apprentice for VLSI design, is perhaps the best known example [96, 81].

[3]We are grateful to Jack Mostow for this observation [98].

However, our approach differs in that we are not necessarily attempting to develop heuristics that make an intractable theory tractable [99, 126]. Rather, the client may simply intend that the user's role become easier as derived generalizations are made available, while the fundamental intractability of the domain remains.

Hill also considers the application of EBG to the domain of program development [62]. However, Hill's research utilizes a first-order encoding, and focuses upon a particular application within formal programming: the generalization of abstract datatype representations. Our work is directed, instead, toward the realization of a common language, $\lambda^{\Box}$Prolog, in which a multiplicity of programming and theorem proving methodologies can be realized.

In contrast with the apprentice approach is that taken by Steier [123]. Steier uses the Soar architecture (see §4.7 & §6.3) to develop a series of algorithm designers that learn from experience. Unlike the work above, his efforts do not focus on cooperative problem solution; rather the system alone constructs programs to meet given criteria using its knowledge-base of design information. And hence it is less critical that the design knowledge within his system be easily comprehended (which may, in part, explain his success employing first-order encodings). At the same time, the programs his framework synthesizes (*e.g.*, sorting algorithms) are significantly more complex than anything to which we have thus far applied our framework, and his learners exhibit improved performance as they encounter similar design problems.

Recently, Hagiya has formalized higher-order EBG over another higher-order language LF [54]. LF, which stands for 'logical framework', is a logic for encoding other logics [60]. By formulating EBG over LF, Hagiya realizes EBG over languages defined in LF. Like ours, his formulation is defined in terms of higher-order unification, but he also extends the algorithm to treat mathematical induction. Hagiya also uses LF and higher-order unification to explore the derivation of programs and proofs by example [56]. Previously Hagiya has presented a solution for generalizing programs (*e.g.*, to operate on greater ranges of input values) in the proofs-as-programs framework using higher-order type theory [55].

```
!! insert_lamC
        (C fix λf. lam λn. G f n)
        (C lam λn. appl (fix λf. lam λn'. G f n') n).
!! add_oper_right_id_1 op C
                (C λx. Gx)
                (C λx. op (G x) a)
   ⇐ right_identity op a.
!! abstract_arg op C1 C2
              (C1 (C2 a))
              (C1 (appl (lam λm. C2 m) a)).
!! name_function C
              (C G)
              (C fix λf. G).
!! unfold C
        (C fix λf. G f)
        (C (G fix λf. G f)).
!! reduce_1 C
          (C λx. appl (lam λn. G n) x)
          (C λx. G x).
!! distribute_if_2 op C
                (C λx. λy. op (if (B x y)
                                  (E1 x y)
                                  (E2 x y))
                              (H x y))
                (C λx. λy. if (B x y)
                              (op (E1 x y) (H x y))
                              (op (E2 x y) (H x y))).
!! left_identity_2 op C
                (C λx. λy. op a (H x y))
                (C λx. λy. H x y)
   ⇐ left_identity op a.
!! reassociate_2 op C
                (C λx. λy. op (op (H1 x y) (H2 x y)) (H3 x y))
                (C λx. λy. op (H1 x y) (op (H2 x y) (H3 x y)))
   ⇐ associative op.
!! fold_two_3 C1;C2 C3 (C2 fix λf. lam λm. lam λn. C3 G n m)
                       (C1 λf. λx. λy. C3 G (H1 x y) (H2 x y))
                       (C1 λf. λx. λy. appl (appl f (H2 x y)) (H2 x y)).
```

Figure 7.1: Transformation Rules

!! **tail_rec** $op\ F_0\ F_{10} \Leftarrow$

insert_lam	$(\lambda C.\, C)$ $F_0\ F_1,$
add_oper_right_id_1	op $(\lambda C.\, \textbf{lam}\ \lambda n.\, C\ n)$ $F_1\ F_2,$
abstract_arg	op $(\lambda C.\, C)$ $(\lambda C.\, \textbf{lam}\ \lambda n.\, op\ (W_0\ n)\ C)$ $F_2\ F_3,$
name_function	$(\lambda C.\, \textbf{appl}\ C\ W)$ $F_3\ F_4,$
unfold	$(\lambda C.\, \textbf{appl}\ (\textbf{fix}\ \lambda f'.\, \textbf{lam}\ \lambda m.\, \textbf{lam}\ \lambda n.\, op\ (\textbf{appl}\ C\ n)\ m)\ W)$ $F_4\ F_5,$
reduce_1	$(\lambda C.\, \textbf{appl}\ (\textbf{fix}\ \lambda f'.\, \textbf{lam}\ \lambda m.\, \textbf{lam}\ \lambda n.\, op\ (C\ n)\ m)\ W)$ $F_5\ F_6,$
distribute_if_2	op $(\lambda C.\, \textbf{appl}\ (\textbf{fix}\ \lambda f'.\, \textbf{lam}\ \lambda m.\, \textbf{lam}\ \lambda n.\, C\ m\ n)\ W)$ $F_6\ F_7,$
left_identity_2	op $(\lambda C.\, \textbf{appl}\ (\textbf{fix}\ \lambda f'.\, \textbf{lam}\ \lambda m.\, \textbf{lam}\ \lambda n.\, \textbf{if}\ (W_1\ m\ n)\ (C\ m\ n)\ (W_3\ m\ n))\ W)$ $F_7\ F_8,$
reassociate_2	op $(\lambda C.\, \textbf{appl}\ (\textbf{fix}\ \lambda f'.\, \textbf{lam}\ \lambda m.\, \textbf{lam}\ \lambda n.\, \textbf{if}\ (W_1\ m\ n)\ (W_2\ m\ n)\ (C\ m\ n))\ W)$ $F_8\ F_9,$
fold_two_3	$(\lambda C.\, \textbf{appl}\ (\textbf{fix}\ \lambda f'.\, \textbf{lam}\ \lambda m.\, \textbf{lam}\ \lambda n.\, \textbf{if}\ (W_1\ m\ n)\ (W_2\ m\ n)\ (C\ f'\ m\ n))\ W)$ $(\lambda C.\, \textbf{appl}\ C\ W)$ $(\lambda C.\, \lambda H_1.\, \lambda H_2.\, op\ (\textbf{appl}\ C\ H_1\ )\ H_2)$ $F_4\ F_9\ F_{10}.$

Figure 7.2: Meta-program

Chapter 8

$\lambda^{\Box}$Prolog and EBG

Within this chapter, we more formally develop the $\lambda^{\Box}$Prolog language and the higher-order EBG algorithm it admits. To that end, we first extend the inference system of §3.7 to realize $\lambda^{\Box}$Prolog. Next, we introduce prototype implementations of $\lambda^{\Box}$Prolog, and then of EBG over $\lambda^{\Box}$Prolog, each through an interpreter written in λProlog. While these interpreters are too slow to be of great practical value, they serve as an abstract specification of both the $\lambda^{\Box}$Prolog logic and the EBG algorithm.

The concepts developed herein (in particular, the λProlog interpreters) are sufficiently detailed and deep that readers will likely have to invest some time studying the presentation (and scrutinizing the code). The more casual reviewer may wish to skim this chapter instead.

Other work. del Cerro offers another approach to incorporating modal logic within the logic programming framework that has nothing to do with EBG [26, 27]. For treatments of automated theorem proving in modal logics outside of logic programming (and EBG), see Wallen [131] and Thistlewaite [128].

8.1 The Logic of $\lambda^{\Box}$Prolog

The syntax of $\lambda^{\Box}$Prolog is summarized by the following inductively defined classes:

$$
\begin{array}{rcl}
G & ::= & \mathbf{true} \mid A \mid G_1 \,,\, G_2 \mid G_1 \,;\, G_2 \mid D \Rightarrow G \mid \forall x\,[:\tau].\, G \mid \exists x\,[:\tau].\, G \mid \Box\, G_{\Box} \\
G_{\Box} & ::= & \mathbf{true} \mid A \mid G_{\Box 1} \,,\, G_{\Box 2} \mid \forall x\,[:\tau].\, G_{\Box} \mid \Box\, G_{\Box} \\
D & ::= & \mathbf{true} \mid A \mid D_1 \,,\, D_2 \mid D \Leftarrow G \mid \forall x\,[:\tau].\, D \mid \Box\, D \\
\mathcal{P} & ::= & \epsilon \mid D.\ \mathcal{P} \mid !!\, D.\ \mathcal{P}
\end{array}
$$

where the new meta-variable $G_{\Box}$ over ranges over 'boxed' goals, and ϵ is the null terminal. Although our examples have mainly employed $\Box$ at the top-level, the above definition points out that $\Box$ is in no way restricted to outermost occurrences.

$\lambda^{\Box}$Prolog does not distinguish sequences of the modal prefix; that is, $\Box\Box M$ is equivalent to $\Box M$. For readers familiar with modal logic, in this respect $\lambda^{\Box}$Prolog may be considered an intuitionistic version of the classical modal logic *S5* [16]. However, $\lambda^{\Box}$Prolog is properly contained within *S5* as it lacks the logical connectives of negation ($\neg$) and the second modal operator of *possibility* $\Diamond$, which may be defined as $\neg\Box\neg$. (The difference between possible and contingent truth is similar to that between contingency and necessity: $\Diamond A$ is to A as A is to $\Box A$. $\lambda^{\Box}$Prolog could equally have been formulated with unprefixed clauses representing domain theory and clauses prefixed with $\Diamond$ standing for training theory.)

The above definition disallows goals of the form $\Box(D \Rightarrow G)$, $\Box(\exists x.G)$, and $\Box(G_1 \,;\, G_2)$. For $\Box(D \Rightarrow G)$, this restriction is motivated by the lack of modally correct strategy (*i.e.*, inference rules) for solving this goal. One possible tactic would be the inference rule

$$\frac{\{\Box\, D\} \cup \mathcal{P} \vdash_\theta G}{\mathcal{P} \vdash_\theta \Box\,(D \Rightarrow G)}$$

Note, however, that the above permits $\Vdash \Box\,(\Box A \Rightarrow A)$ to succeed, although it is clearly an invalid goal. For the remaining disjunctive and existential G-forms — $\Box(G_1 \,;\, G_2)$ and $\Box(\exists x.G)$ — there exist valid inference rules:

$$\frac{\mathcal{P} \vdash_\theta \Box\, G_1}{\mathcal{P} \vdash_\theta \Box\,(G_1 \,;\, G_2)}$$

$$\frac{\mathcal{P} \vdash_\theta \Box\, G_2}{\mathcal{P} \vdash_\theta \Box\,(G_1 \,;\, G_2)}$$

$$\frac{\mathcal{P} \vdash_\theta \Box\, Gy}{\mathcal{P} \vdash_{\theta\backslash y} \Box\,(\exists x{:}\tau.\, Gx)} \qquad \text{where } y \notin \mathsf{free}(G).$$

However, we question the usefulness of the above inferences, and whether any additional expressivity is provided by those G-forms. For now, we have made the simplifying assumption of disallowing each of these goals.

8.2 Normal-form for Clauses

The λProlog inference system of §3.7 may be extended to realize 'pure' $\lambda^{\Box}$Prolog. (By 'pure' we simply mean the logical foundation of the language — that is, the logical connectives without EBG, **rule**, '!', *etc.*) Before presenting that interpretation, we first derive a normal-form for arbitrary $\lambda^{\Box}$Prolog D-forms, analogous to that given for λProlog in §3.4.2. This normal-form is exploited by the inference system of §8.3 as well as our full $\lambda^{\Box}$Prolog implementation (Chapter 9).

The new normal-form D_{nf} is defined as

$$\begin{array}{rcl} D_{\mathsf{nf}} & ::= & D_{\mathsf{nf}} \,,\, D_{\mathsf{nf}} \;\mid\; D_{\forall} \\ D_{\forall} & ::= & \forall x.\, D_{\forall} \;\mid\; A \Leftarrow G^{\mathbf{w}} \;\mid\; (\Box\, D_{\Box\forall}) \Leftarrow G^{\mathbf{w}} \\ D_{\Box\forall} & ::= & \forall y.\, D_{\Box\forall} \;\mid\; A \Leftarrow G^{\mathbf{s}} \end{array}$$

Under this definition, the program $\mathcal{P}$ is mapped to a set of clauses, each of which is either of the previous normal-form —

$$\forall \mathcal{W}.\, A \Leftarrow G^{\mathbf{w}}.$$

or of the extended normal-form —

$$\forall \mathcal{W}.\, (\Box\, \forall \mathcal{S}.\, A \Leftarrow G^{\mathbf{s}}) \Leftarrow G^{\mathbf{w}}.$$

where variables of $\mathcal{W}$ may appear free in $G^{\mathbf{w}}$, and those of both $\mathcal{W}$ and $\mathcal{S}$ may appear free in A and $G^{\mathbf{s}}$. (We take the liberty of dropping the $\mathcal{W}$ and $\mathcal{S}$ on A as the alternative $A_{\mathcal{WS}}$ becomes overly intrusive.) As before, an atomic clause A becomes $A \Leftarrow \mathbf{true}$, but now a 'boxed' atomic clause $\Box\, A$ becomes $\Box\, (A \Leftarrow \mathbf{true}) \Leftarrow \mathbf{true}$.

The validity of D_{nf} relies upon our decision to collapse sequences of the Modal operator — $\Box\Box D$ goes to $\Box D$. To illustrate, the D-forms on the left are mapped to the normal-D-forms on the right:

$\mathbf{q}$	$\mathbf{q} \Leftarrow \mathbf{true}$
$\Box\, \mathbf{q}$	$\Box\, (\mathbf{q} \Leftarrow \mathbf{true}) \Leftarrow \mathbf{true}$
$\forall x.\, \Box\, \mathbf{q}\, x$	$\forall x.\, (\Box\, (\mathbf{q}\, x \Leftarrow \mathbf{true}) \Leftarrow \mathbf{true})$
$\Box\, \forall x.\, \mathbf{q}\, x$	$(\Box\, \forall x.\, \mathbf{q}\, x \Leftarrow \mathbf{true}) \Leftarrow \mathbf{true}$
$\Box\, (\mathbf{p} \Rightarrow \Box\, (\mathbf{r} \Rightarrow \mathbf{q}))$	$\Box\, (\mathbf{q} \Leftarrow (\mathbf{p}, \mathbf{r})) \Leftarrow \mathbf{true}$
$\Box\, ((\Box\, \mathbf{p}, \mathbf{r}) \Rightarrow (\Box\, \mathbf{q}, \mathbf{s}))$	$\Box\, (\mathbf{q} \Leftarrow (\Box\, \mathbf{p}, \mathbf{r})) \Leftarrow \mathbf{true}$, $\Box\, (\mathbf{s} \Leftarrow (\Box\, \mathbf{p}, \mathbf{r})) \Leftarrow \mathbf{true}$
$\Box\, (\Box\, \mathbf{p} \Rightarrow \Box\, (\Box\, \mathbf{r} \Rightarrow \Box\, \mathbf{q}))$	$\Box\, (\mathbf{q} \Leftarrow (\Box\, \mathbf{p}, \Box\, \mathbf{r})) \Leftarrow \mathbf{true}$
$\mathbf{p} \Rightarrow \forall x.\, \Box\, (\mathbf{s}, (\mathbf{r}\, x \Rightarrow \mathbf{q}))$	$\forall x.\, (\Box\, (\mathbf{q} \Leftarrow \mathbf{r}\, x) \Leftarrow \mathbf{p})$, $\forall x.\, (\Box\, (\mathbf{s} \Leftarrow \mathbf{true}) \Leftarrow \mathbf{p})$

In §3.6 we developed a λProlog program mapping arbitrary λProlog D-forms to a normal-form. In Figures 8.1 & 8.2, we give an analogous mapping (again within λProlog) for $\lambda^{\Box}$Prolog D-forms. As before, the predicate **requantify** moves quantifiers down; the predicate **conjoin** collects preconditions; and the predicate **ndform** coordinates the other predicates. Now, however, each of these predicates has a *weak* (prefixed by '**w**') and a *strong* (prefixed by '**s**') version. The '**s**' is indicative of those components that are nested under a $\Box$ within the original D-form, while the '**w**' is for components not so nested. In fact, the new normal-form is most easily viewed as two levels of the preceding normal-form: one for the 'boxed' portion; the other, for the 'unboxed' (although both $G^{\mathbf{w}}$ and $G^{\mathbf{s}}$ may themselves contain $\Box$).

$$
\begin{array}{llll}
\textbf{wrequantify} & (\forall x.\, D_1 x\ ,\ D_2 x) & (D_1'\ ,\ D_2') \Leftarrow\ !, & \textbf{wrequantify}\ (\forall\ D_1)\ D_1', \\
 & & & \textbf{wrequantify}\ (\forall\ D_2)\ D_2' \\
\textbf{wrequantify} & D & D. & \\
 & & & \\
\textbf{srequantify} & (\forall x.\, D_1 x\ ,\ D_2 x) & (D_1'\ ,\ D_2') \Leftarrow\ !, & \textbf{srequantify}\ (\forall\ D_1)\ D_1', \\
 & & & \textbf{srequantify}\ (\forall\ D_2)\ D_2' \\
\textbf{srequantify} & (\forall x.\, \Box\ Dx) & (\Box\ D') \Leftarrow\ !, & \textbf{srequantify}\ (\forall x.\, Dx)\ D' \\
\textbf{srequantify} & D & D. &
\end{array}
$$

$$
\begin{array}{llll}
\textbf{wconjoin} & G\ (D_1\ ,\ D_2) & (D_1'\ ,\ D_2') \Leftarrow & \textbf{wconjoin}\ G\ D_1\ D_1', \\
 & & & \textbf{wconjoin}\ G\ D_2\ D_2'. \\
\textbf{wconjoin} & G\ (\forall\ D) & (\forall\ D') \Leftarrow & \forall x.\, \textbf{wconjoin}\ G\ (Dx)\ (D'x). \\
\textbf{wconjoin} & G\ (A \Leftarrow \textbf{true}) & (A \Leftarrow G). & \\
\textbf{wconjoin} & G\ (A \Leftarrow G_1) & (A \Leftarrow (G\ ,\ G_1)). &
\end{array}
$$

$$
\begin{array}{llll}
\textbf{sconjoin} & G\ (D_1\ ,\ D_2) & (D_1'\ ,\ D_2') \Leftarrow & \textbf{sconjoin}\ G\ D_1\ D_1', \\
 & & & \textbf{sconjoin}\ G\ D_2\ D_2'. \\
\textbf{sconjoin} & G\ (\forall\ D) & (\forall\ D') \Leftarrow & \forall x.\, \textbf{sconjoin}\ G\ (Dx)\ (D'x). \\
\textbf{sconjoin} & G\ (\Box\ D) & (\Box\ D') \Leftarrow & \textbf{sconjoin}\ G\ D\ D'. \\
\textbf{sconjoin} & G\ (A \Leftarrow \textbf{true}) & (A \Leftarrow G). & \\
\textbf{sconjoin} & G\ (A \Leftarrow G_1) & (A \Leftarrow (G\ ,\ G_1)). &
\end{array}
$$

Figure 8.1: Clause normal-form conversion (Part 1).

```
wndform (D1 , D2)  (D1' , D2')   ⇐ !, wndform D1 D1',
                                    wndform D2 D2'.
wndform (∀ D)      D''           ⇐ !, (∀x. wndform (Dx) (D'x)),
                                    wrequantify (∀ D') D''.
wndform (D ⇐ G)    D''           ⇐ !, ndform D D',
                                    wconjoin G D' D''.
wndform (□ D)      D''           ⇐ !, sndform D D',
                                    conjoin_true D' D''.
wndform A          (A ⇐ true).

sndform (D1 , D2)  (D1' , D2')   ⇐ !, sndform D1 D1',
                                    sndform D2 D2'.
sndform (∀ D)      D''           ⇐ !, (∀x. sndform (Dx) (D'x)),
                                    srequantify (∀ D') D''.
sndform (D ⇐ G)    D''           ⇐ !, sndform D D',
                                    sconjoin G D' D''.
sndform (□ D)      D'            ⇐ !, sndform D D'.
sndform A          (□ (A ⇐ true)).

conjoin_true (D1 , D2)  (D1' , D2')   ⇐ conjoin_true D1 D1',
                                        conjoin_true D2 D2'.
conjoin_true (□ D)      (□ D ⇐ true).

ndform D D' ⇐ wndform D D'.
```

Figure 8.2: Clause normal-form conversion (Part 2).

$$\frac{\mathcal{P} \vdash^{\mathsf{s}}_{\theta} G}{\mathcal{P} \vdash^{\mathsf{w}}_{\theta} (\Box\, G)}$$

$$\frac{(\forall \mathcal{W}.\, A \Leftarrow G^{\mathsf{w}}) \in \mathsf{nf}(\mathcal{P}) \qquad \theta\sigma_{\mathcal{W}} A = \theta A' \qquad \theta\mathcal{P} \vdash^{\mathsf{w}}_{\psi} \theta\sigma_{\mathcal{W}} G^{\mathsf{w}}}{\mathcal{P} \vdash^{\mathsf{w}}_{\psi\theta} A'}$$

where $\mathsf{dom}(\sigma_{\mathcal{W}}) \subseteq \mathcal{W}$,
$\mathsf{dom}(\theta) \cap \mathcal{W} = \emptyset$,
and $\sigma_{\mathcal{W}}$ & θ are minimal.

$$\frac{(\forall \mathcal{W}.\, (\Box\, \forall \mathcal{S}.\, A \Leftarrow G^{\mathsf{s}}) \Leftarrow G^{\mathsf{w}}) \in \mathsf{nf}(\mathcal{P}) \qquad \theta\sigma_{\mathcal{W}}\sigma_{\mathcal{S}} A = \theta A' \qquad \theta\mathcal{P} \vdash^{\mathsf{w}}_{\psi} \theta\sigma_{\mathcal{W}} G^{\mathsf{w}}\,,\, \theta\sigma_{\mathcal{W}}\sigma_{\mathcal{S}} G^{\mathsf{s}}}{\mathcal{P} \vdash^{\mathsf{w}}_{\psi\theta} A'}$$

where $\mathsf{dom}(\sigma_{\mathcal{W}}) \subseteq \mathcal{W}$,
$\mathsf{dom}(\sigma_{\mathcal{S}}) \subseteq \mathcal{S}$,
$\mathsf{dom}(\theta) \cap \mathcal{W} = \emptyset$, $\mathsf{dom}(\theta) \cap \mathcal{S} = \emptyset$,
and $\sigma_{\mathcal{W}}$, $\sigma_{\mathcal{S}}$ & θ are minimal.

Figure 8.3: Partial 'weak' inference rules for $\lambda^{\Box}$Prolog.

One use of higher-order matching exploited within Figures 8.1 & 8.2 deserves further explanation: By matching D against the λ-term $(\forall x.\, D_1\, x\,,\, D_2\, x)$, we insure that D is a universally quantified conjunction, and that D_1 and D_2 are bound to the appropriate functions of x within D. For example, for $D = (\forall x.\, \mathbf{a}\, x\,,\, \mathbf{b}\, x)$, the preceding instantiates $D_1 = \mathbf{lam}\ \lambda x.\, \mathbf{a}\, x$ and $D_2 = \mathbf{lam}\ \lambda x.\, \mathbf{b}\, x$. Similarly, for $D = \forall x.\, \mathbf{a}\, x$, unifying D with $\forall\, D_1$ instantiates $D_1 = \mathbf{lam}\ \lambda x.\, \mathbf{a}\, x$.

We do not herein attempt to formally establish that all $\lambda^{\Box}$Prolog D-forms can be mapped to D_{nf}, although the code provides some evidence for this. Nor, for that matter, do we argue further that the mapping to D_{nf} is meaning-preserving. A proof could take the form of an extended set of distributive transformations analogous to those presented in §3.6.

8.3 Inference System for $\lambda^{\Box}$Prolog.

It is important to distinguish the programming language $\lambda^{\Box}$Prolog from the process that produces explanation-based generalizations of $\lambda^{\Box}$Prolog computation. Within this section we further develop the former by extending the inference system of §3.7 to implement pure $\lambda^{\Box}$Prolog. To that end, we split the $\vdash$ relation into $\vdash^{\mathsf{w}}$ and $\vdash^{\mathsf{s}}$ — the former for the derivation of unboxed ('weak') goals; the latter for that of boxed ('strong'): for example, from the clause $\mathbf{p}$ we cannot derive the goal $\Box\, \mathbf{p}$, but the goal $\mathbf{p}$ does follow from the clause $\Box\, \mathbf{p}$. Initial queries then are phrased as $\vdash^{\mathsf{w}} G$, but the inference $\vdash^{\mathsf{w}} \Box\, G$ is defined in terms of $\vdash^{\mathsf{s}} G$.

$$\frac{}{\mathcal{P} \vdash^{s} \mathbf{true}}$$

$$\frac{\mathcal{P} \vdash^{s}_{\theta} G_1 \qquad \theta\mathcal{P} \vdash^{s}_{\psi} \theta G_2}{\mathcal{P} \vdash^{s}_{\theta\psi} (G_1 \,,\, G_2)}$$

$$\frac{\mathcal{P} \vdash^{s}_{\theta} Gy}{\mathcal{P} \vdash^{s}_{\theta} (\forall x{:}\tau.\, Gx)} \qquad \text{where } y \notin \mathsf{free}(\theta G),\ y \notin \mathsf{free}(\theta\mathcal{P}), \text{ and } y \notin \mathsf{dom}(\theta)$$

$$\frac{\mathcal{P} \vdash^{s}_{\theta} G}{\mathcal{P} \vdash^{s}_{\theta} (\Box\, G)}$$

$$\frac{(\forall \mathcal{W}.\, (\Box\, \forall \mathcal{S}.\, A \Leftarrow G^{\mathbf{s}}) \Leftarrow G^{\mathbf{w}}) \in \mathsf{nf}(\mathcal{P}) \qquad \theta\sigma_{\mathcal{W}}\sigma_{\mathcal{S}} A = \theta A' \qquad \theta\mathcal{P} \Vdash^{s}_{\psi}\ \theta\sigma_{\mathcal{W}} G^{\mathbf{w}},\ \Box\, \theta\sigma_{\mathcal{W}}\sigma_{\mathcal{S}} G^{\mathbf{s}}}{\mathcal{P} \vdash^{s}_{\psi\theta} A'}$$

where $\mathsf{dom}(\sigma_{\mathcal{W}}) \subseteq \mathcal{W}$,
$\mathsf{dom}(\sigma_{\mathcal{S}}) \subseteq \mathcal{S}$,
$\mathsf{dom}(\theta) \cap \mathcal{W} = \emptyset$, $\mathsf{dom}(\theta) \cap \mathcal{S} = \emptyset$,
and $\sigma_{\mathcal{W}}$, $\sigma_{\mathcal{S}}$ & θ are minimal.

Figure 8.4: 'Strong' inference rules for $\lambda^{\Box}$Prolog.

As one might expect, $\vdash^{\mathbf{w}}$ largely follows the definition of $\vdash$: In Figure 8.3 we list only those rules that have changed. For completeness, we include the full definition of $\Vdash^{s}$ in Figure 8.4, although it too largely follows $\vdash$; in fact, only the final three rules differ. Of particular importance is that only D-forms containing $\Box$ are used to establish $\vdash^{s} G$: we ensure the necessary truth of G by requiring that each to its deriving clauses is also necessarily true.

Informal $\lambda^{\Box}$Prolog interpreter. For those readers preferring a 'logic programming' oriented description of the interpretation of $\lambda^{\Box}$Prolog, we offer the following insight into the preceding inference rules. Subsequent sections will further explicate $\lambda^{\Box}$Prolog.

As within our informal λProlog interpreter of §3.4.2, $\lambda^{\Box}$Prolog goals are herein reduced to a atomic subgoal $G_{\mathbf{a}}$ for solution. And as within the inference rules, necessary (boxed) goals are strongly solved; contingent ones, weakly solved. For strong solution, applied clauses must be of the extended normal form (*i.e.*, contain $\Box$), since only a necessarily true clause can establish a necessarily true goal. On the other hand, clauses of either normal-form are relevant for weak solution.

From $G_{\mathbf{a}}$ and the normal D-form

$$\forall \mathcal{W}.\, (\Box\, \forall \mathcal{S}.\, A \Leftarrow G^{\mathbf{s}}) \Leftarrow G^{\mathbf{w}}.$$

(unboxed D-forms are handled as before), we precede as follows:

- Create new logical variables $\hat{\mathcal{W}}$ and $\hat{\mathcal{S}}$ for each universal variable in $\mathcal{W}$ and $\mathcal{S}$, and then substitute $\hat{\mathcal{W}}$ for $\mathcal{W}$ and $\hat{\mathcal{S}}$ for $\mathcal{S}$ in A, $G^{\mathbf{w}}$, and $G^{\mathbf{s}}$, yielding $\hat{A}$, $\hat{G^{\mathbf{w}}}$, and $\hat{G^{\mathbf{s}}}$, respectively.
- Unify $\hat{A}$ and $G_{\mathbf{a}}$.
- For strong solution, solve ($\hat{G^{\mathbf{w}}}$, $\Box$ $\hat{G^{\mathbf{s}}}$). For weak solution, solve ($\hat{G^{\mathbf{w}}}$, $\hat{G^{\mathbf{s}}}$).

8.4 Introduction to the Meta-Interpreters

While the preceding inference system provides a formal characterization of the $\lambda^{\Box}$Prolog logic, we have yet to treat higher-order EBG. To that end, we develop a program that implements EBG over $\lambda^{\Box}$Prolog computation. This implementation consists of an extended $\lambda^{\Box}$Prolog interpreter written in λProlog. To simplify discussion, we first present, in §8.5, the basic $\lambda^{\Box}$Prolog interpreter without the generalizing component. This $\lambda^{\Box}$Prolog interpreter provides a formal operational specification of $\lambda^{\Box}$Prolog which, for the most part, mirrors that given by the inference system of §8.3. Due to the closeness of the correspondence between the object-language ($\lambda^{\Box}$Prolog) and the meta-language (λProlog), we shall often use the more descriptive term 'meta-interpreter.'

Our $\lambda^{\Box}$Prolog meta-interpreter is extended to perform EBG within a second prototype in §8.6. This expanded meta-interpreter exemplifies the generalization algorithm, and has reproduced most of the examples contained within this dissertation. (Others were derived under the full implementation described in Chapter 9.) So that our presentation is more accessible, we have deferred some less pertinent details of the generalizing meta-interpreter to Appendix A.4.

Finally, in §8.7 we further extend this meta-interpreter to admit operationality criteria.

We chose to prototype $\lambda^{\Box}$Prolog and higher-order EBG in this manner to facilitate experimentation with alternative formulations of both the language and the generalization algorithm, and moreover, to provide a formal specification of each. The prototypes are sufficiently slow and limited, however, that a more direct implementation was eventually required (Chapter 9).

8.4.1 Accessing the Logic Program

To run examples under the meta-interpreters to follow, the $\lambda^{\Box}$Prolog program $\mathcal{P}_{ob}$ to be interpreted must be available as data. This is accomplished by assuming **hyp** D for each clause D of $\mathcal{P}_{ob}$ prior to invoking the meta-interpreter. **hyp** addresses the need for *reification* — the mapping from program to data. (Reification is the inverse of reflection; see §5.3.1.) Within Prolog reification is accomplished with **clause** D G, which matches against clauses in the program-base, instantiating D to the head and G to the corresponding body. λProlog

```
wsolve  true        ⇐  !.
wsolve  (G1 , G2)   ⇐  !, wsolve G1, wsolve G2.
wsolve  (G1 ; G2)   ⇐  !, (wsolve G1; wsolve G2).
wsolve  (D ⇒ G)     ⇐  !, hyp D ⇒ wsolve G.
wsolve  (∀ G)       ⇐  !, ∀x. wsolve (G x).
wsolve  (∃ G)       ⇐  !, ∃t. wsolve (G t).
wsolve  (□ G)       ⇐  !, ssolve G.
wsolve  Ga          ⇐  !, hyp D, wmatch D Ga Gs, wsolve Gs.

ssolve  true        ⇐  !.
ssolve  (G1 , G2)   ⇐  !, ssolve G1, ssolve G2.
ssolve  (∀ G)       ⇐  !, ∀x. ssolve (G x).
ssolve  (□ G)       ⇐  !, ssolve G.
ssolve  Ga          ⇐  !, hyp D, smatch D Ga Gs, wsolve Gs.
```

Figure 8.5: Meta-interpreter without EBG: Goal analysis.

---

does *not* provide a **clause** construct.[1] Hence in λProlog, to manipulate programs and then run the derived results directly requires that two versions of the program be present: the initial one $\mathcal{P}_{ob}$, which is available as data (via an indexing predicate such as **hyp**), and the reflected one added to the logic program $\mathcal{P}$.

hyp allows the meta-interpreter to enumerate the program $\mathcal{P}_{ob}$ to be interpreted with λProlog's backtracking search (by successively solving the goal **hyp** D), although obviously the performance of such an approach suffers in comparison with the schemes employed by more standard logic programming implementations (*e.g.*, hashing on the name of the predicate heading an atom).

As mentioned above, the variables of clauses asserted with **hyp** must be explicitly universally quantified: the scope of λProlog's implicit quantification is insufficient as it is includes **hyp** as well. (The '!!' convention, while part of the eventual system, only functions at the top-level, and moreover, is not realizable within the prototype interpreter.) What follows is a portion of the ubiquitous suicide example in the form recognized by the meta-interpreter:

```
hyp (□ ∀a. ∀b. ∀c.  kill a b  ⇐  hate a b, possess a c, weapon c).
hyp (gun obj1).
```

8.5 The Meta-Interpreter

Our λ$^{\square}$Prolog interpreter is divided between two sets of clauses: the **solve** predicates of Figure 8.5, which reduce a given λ$^{\square}$Prolog goal G to some number of atomic subgoals (G_a's),

---

[1] A λProlog **clause** would simply take the form **clause** D, but would also be more complex in that it presumably would explicitly quantify universal variables. There does not appear, however, to be any logical problem with adding **clause** to λProlog.

and the **match** predicates of Figure 8.6, which attempt to derive a pending atomic subgoal G_a from the program $\mathcal{P}_{ob}$.

The goal reduction performed by **solve** is again split between two sets of clauses: **wsolve** for 'weak-solve' and **ssolve** for 'strong-solve.' This distinction, analogous to that made between $\Vdash^w$ and $\Vdash^s$, arises from the more stringent proof required by the necessary truth of 'boxed' goals: $\{\Box\, \mathbf{p}\} \vdash \mathbf{p}$, but *not* $\{\mathbf{p}\} \vdash \Box : \mathbf{p}$. The top-level predicate is **wsolve**, because goals are contingent until a $\Box$ has been encountered. Each of the G_a's derived through **solve** will require either a 'strong' or 'weak' proof, which is realized through the corresponding **match** predicates — **wmatch** and **smatch**.

Within the **solve** predicates, the solution of a $\lambda^{\Box}$Prolog goal is largely realized by the corresponding λProlog construct. For example, a $\lambda^{\Box}$Prolog conjunction (G_1 , G_2) is derived by establishing the λProlog conjunction of the solutions to G_1 and G_2. Similarly, a universally quantified $\lambda^{\Box}$Prolog goal is universally derived under λProlog. And an implicational goal $D \Rightarrow G$ is proven by first assuming D, and then attempting to derive G. Such sharing between object-language ($\lambda^{\Box}$Prolog) and meta-language (λProlog) makes for elegant interpretation. (The rules of **ssolve** do not address the range of λProlog connectives because of the additional restrictions placed upon boxed goals; see §8.1.)

In the final clauses of **wsolve** and **ssolve**, the pending goal has been reduced to an atomic G_a. This is insured by our use of the cut operator '!' described in §3.5. Cut's only effect within **solve** is to insure that G_a is indeed atomic: if G_a instead contained a logical connective, '!' would not have permitted the interpretation to 'fall through' to its present position, as one of the preceding clauses would have been chosen.

Through the predicate **hyp**, the final clauses of **wsolve** and **ssolve** select a potentially pertinent clause D from the program, which the **match** predicates then attempt to apply in the proof of G_a. The selection of D is inefficient in that each clause of $\mathcal{P}_{ob}$ is simply tried in order until one is found that derives G_a. (For the purposes of this meta-interpretation, D-forms need *not* be in normal-form.) As we shall see, in the course of deriving G_a from D, **match** may produce subgoals (G_s's) that must be subsequently solved to complete the proof.

The **match** predicates analyze the selected program clause D to determine if it is applicable in the solution of G_a. For a conjunction (D_1 , D_2), the (intuitionistic) logic programming paradigm dictates that either D_1 or D_2 individually derives G_a (although both D_1 and D_2 are available for the derivation of any resulting subgoals.) A universally quantified clause $\forall\, D$ (or equivalently, $\forall x.Dx$) is reduced by replacing the bound variable with a new logical variable Y, which may become instantiated in the course of the proof: for example, the clause $\forall z.$ **weapon** $z \Leftarrow$ **gun** z becomes **weapon** $Y \Leftarrow$ **gun** Y. If D is a rule $D' \Leftarrow G'$, we conjoin G' with the subgoals that arise from establishing that D' implies G_a: for the clause **weapon** $Y \Leftarrow$ **gun** Y, the interpreter first determines whether **weapon** Y establishes G_a, and then attempts to solve **gun** Z. When **smatch** encounters a $\Box$ in the program, the nested clause need only be weakly matched with the current goal. This is because proving a goal 'strongly' simply requires that any utilized clauses must themselves be necessarily true. The resulting subgoal G_s is, however, boxed as it too must be strongly proved. On the other

```
wmatch  (D1 , D2)   Ga   Gs        ⇐ !, (wmatch D1 Ga Gs; wmatch D2 Ga Gs).
wmatch  (D ⇐ G)     Ga   (G , Gs)  ⇐ !, wmatch D Ga Gs.
wmatch  (∀ D)       Ga   Gs        ⇐ !, wmatch (D y) Ga Gs.
wmatch  (□ D)       Ga   Gs        ⇐ !, wmatch D Ga Gs.
wmatch  Da          Ga   true      ⇐ !, Da = Ga.

smatch  (D1 , D2)   Ga   Gs        ⇐ !, (smatch D1 Ga Gs; smatch D2 Ga Gs).
smatch  (D ⇐ G)     Ga   (G , Gs)  ⇐ !, smatch D Ga Gs.
smatch  (∀ D)       Ga   Gs        ⇐ !, smatch (D y) Ga Gs.
smatch  (□ D)       Ga   (□ Gs)    ⇐ !, wmatch D Ga Gs.
```

Figure 8.6: Meta-interpreter without EBG: Clause analysis.

hand, **wmatch** ignores □'s within D, because therein we are only concerned with a weak proof.

In the final clause of **wmatch**, the unification of an atomic D_a and G_a is attempted: for example, unifying the goal **weapon obj1** with the clause **weapon** Y. This is analogous to the unification of a goal and clause head under a Prolog interpretation. If successful, this has the effect of 'returning' the accumulated conjunction of subgoals G_s (in this case, **gun obj1**) to the last clause of **solve**, which then derives G_s recursively. The predicate **smatch** is, however, missing the analogue to the last clause of **wmatch**. This is because a contingent atomic clause cannot be used to prove a necessary atomic goal; that is the clause **p** is not sufficient to derive □ **p**.

This concludes the discussion of the basic $\lambda^{\square}$Prolog meta-interpreter. The next step is extending it to perform EBG.

8.6 The Generalizing Meta-Interpreter

Within this section we extend the $\lambda^{\square}$Prolog meta-interpreter of §8.5 to perform EBG. This section focuses on developing the most relevant and interesting aspects of the prototype; the unabridged meta-interpreter may be found in Appendix A.4.

Kedar-Cabelli & McCarty produce first-order explanation-based generalizations within Prolog via an augmented meta-interpreter [71]. As we shall take a similar approach, we briefly review Kedar-Cabelli & McCarty's implementation: Under its second formulation (pp. 387–388), their meta-interpreter, **prolog_ebg**, solves a particular query in parallel with the construction of the associated explanation-based generalization. The predicate **prolog_ebg** takes three arguments: the particular query G, the generalized query GG, and the conjunction of generalized conditions DD sufficient to establish GG.

Each 'rule' applied by **prolog_ebg** in the proof of G is similarly applied in the proof of GG. Leaves of the Prolog computation that arise in the course of deriving GG (*i.e.*, those goals

```
wsolve  true          true          true          ⇐ !.
wsolve  (G1 , G2)     (GG1 , GG2)   (DD1 , DD2)   ⇐ !,  wsolve G1 GG1 DD1, wsolve G2 GG2 DD2.
wsolve  (G1 ; G2)     (GG1 ; GG2)   DD            ⇐ !,  (wsolve G1 GG1 DD; wsolve G2 GG2 DD).
wsolve  (D ⇒ G)       GG            DD            ⇐ !,  hyp D ⇒ wsolve G GG DD.
wsolve  (∀ G)         (∀ GG)        (DD x)        ⇐ !,  ∀x. wsolve (G x) (GG x) (DD x).
wsolve  (∃ G)         (∃ GG)        (DD t)        ⇐ !,  wsolve (G t) (GG t) (DD t).
wsolve  (□ G)         (□ GG)        DD            ⇐ !,  ssolve G GG DD.
wsolve  Ga            GGa           (DD1 , DD2)   ⇐ !,  hyp D,
                                                        wmatch D Ga DD1 GGa MG,
                                                        meta_wsolve MG DD2.

ssolve  true          true          true          ⇐ !.
ssolve  (G1 , G2)     (GG1 , GG2)   (DD1 , DD2)   ⇐ !,  ssolve G1 GG1 DD1, ssolve G2 GG2 DD2.
ssolve  (∀ G)         (∀ GG)        (DD x)        ⇐ !,  ∀x. ssolve (G x) (GG x) (DD x).
ssolve  (□ G)         (□ GG)        DD            ⇐ !,  ssolve G GG DD.
ssolve  Ga            GGa           (DD1 , DD2)   ⇐ !,  hyp D,
                                                        smatch D Ga DD1 GGa MG,
                                                        meta_wsolve MG DD2.
```

Figure 8.7: Generalizing meta-interpreter: Goal analysis.

established by 'facts') are accumulated in the conjunction of sufficient conditions *DD*. The resulting explanation-based generalization is then $GG \Leftarrow DD$, where for example

GG = **kill** x x
DD = **depressed** x, **buy** x y, **gun** y

No explicit representation of the proof need be constructed; it is inherent in the Prolog search.

As in the first-order approach of Kedar-Cabelli and McCarty [71], our generalizing meta-interpreter develops two parallel proofs simultaneously: a proof of G and a generalized proof of GG. Again these proofs are not explicitly constructed; rather they are implicit in the λProlog search. In the course of deriving G and GG, the implementation accumulates the conjunction of generalized clauses DD sufficient to establish GG — that is, the leaves of the generalized proof. Figure 8.7 contains the extended **solve** predicates of the generalizing interpreter, which also accept three arguments — the goal G (instantiated), the generalized goal GG (uninstantiated), and the conjunction of generalized sufficient conditions DD (uninstantiated). The resulting explanation-based generalization is then $!!\ GG \Leftarrow DD$.

In the extended **wsolve** and **ssolve**, the decomposition of G guides the corresponding instantiation of the generalized goal GG. It is only at the atomic level where G and GG diverge. (An exception is made for the handling of implicational goals $D' \Rightarrow G'$, which is simplified by locally treating D' as a part of $\mathcal{P}_{obs}$.) The MG's (for 'meta-subgoal') in the final clauses of **solve** assume a role analogous to that played by subgoals in the previous meta-interpreter — that is, MG's retain subproof tasks for later derivation. The transition

```
wmatch  (D1 , D2)   Ga   DD            GGa   MG          ⇐ !,  (wmatch D1 Ga DD GGa MG
                                                                ; wmatch D2 Ga DD GGa MG).
wmatch  (D ⇐ G)     Ga   (DD ⇐ GG)     GGa   (mg G GG,
                                                    MG)   ⇐ !,  wmatch D Ga DD GGa MG.
wmatch  (∀ D)       Ga   DD            GGa   MG          ⇐ !,  wmatch (D x) Ga DD GGa MG.
wmatch  (□ D)       Ga   (□ D)         GGa   MG          ⇐ !,  bmatch D Ga D GGa MG.
wmatch  Ga          Ga   GGa           GGa   true.

smatch  (D1 , D2)   Ga   DD            GGa   MG          ⇐ !,  (smatch D1 Ga DD GGa MG
                                                                ; smatch D2 Ga DD GGa MG).
smatch  (D ⇐ G)     Ga   (DD ⇐ GG)     GGa   (mg G GG,
                                                    MG)   ⇐ !,  smatch D Ga DD GGa MG.
smatch  (∀ D)       Ga   DD            GGa   MG          ⇐ !,  smatch (D x) Ga DD GGa MG.
smatch  (□ D)       Ga   (□ D)         GGa   (□ MG)      ⇐ !,  bmatch D Ga D GGa MG.

bmatch  (D1 , D2)   Ga   (DD1 , DD2)   GGa   MG          ⇐ !,  (bmatch D1 Ga DD1 GGa MG
                                                                ; bmatch D2 Ga DD2 GGa MG).
bmatch  (D ⇐ G)     Ga   (DD ⇐ GG)     GGa   (mg G GG,
                                                    MG)   ⇐ !,  bmatch D Ga DD GGa MG.
bmatch  (∀ D)       Ga   (∀ DD)        GGa   MG          ⇐ !,  bmatch (D x) Ga (DD y) GGa MG.
bmatch  (□ D)       Ga   (□ D)         GGa   MG          ⇐ !,  bmatch D Ga D GGa MG.
bmatch  Ga          Ga   GGa           GGa   true.
```

Figure 8.8: Generalizing meta-interpreter: Clause analysis.

---

from the G_s's of the first interpreter to the current MG's comes out of the need to maintain both G and GG for subsequent solution. The straight-forward clauses **meta_wsolve** and **meta_ssolve** that interpret MG's are given within Figure 8.9.

After **solve** selects a clause D with which to derive G_a, the extended **match** predicates of Figure 8.8, attempt to apply D in the solution of G_a. But in the course of deriving G_a, the new **match** also yields a generalized atomic goal GG_a and a generalized clause DD sufficient to establish GG_a. Within the final clause of **wmatch** where D_a is unified with G_a, DD is instead unified with GG_a. That neither the pair G_a and GG_a nor the pair D_a and DD are unified is essential for generalization: DD and GG need only be instantiated to the point that GG necessarily follows from DD.

How then do any of the constants of D (first or higher-order) ever end up in GG or DD? The answer is that unless some of the D's employed in the proof are boxed, none ever will. In the matching of boxed clauses, D and DD are explicitly unified in the invocation of **bmatch** (for 'boxed-match'): within the suicide problem, for example, both D and DD are bound to $\forall z.$ **weapon** $z \Leftarrow$ **gun** z. (The additional predicate **bmatch** is required to handle subtle differences in the matching of instantiated DD's.) While D and DD are initially equivalent within **bmatch**, they may later diverge as distinct new logical variables x and y are substituted for universally quantified programs. This is because D is to be unified with G_a, while DD is to be unified with GG_a: again for the **weapon** clause, D's logical variable

```
meta_wsolve  true          true          ⇐ !.
meta_wsolve  (MG1 , MG2)   (DD1 , DD2)   ⇐ !,  meta_wsolve MG1 DD1,
                                               meta_wsolve MG2 DD2.
meta_wsolve  (□ MG)        DD            ⇐ !,  meta_ssolve MG DD.
meta_wsolve  (mg G GG)     DD            ⇐ !,  wsolve G GG DD.

meta_ssolve  true          true          ⇐ !.
meta_ssolve  (MG1 , MG2)   (DD1 , DD2)   ⇐ !,  meta_ssolve MG1 DD1,
                                               meta_ssolve MG2 DD2.
meta_ssolve  (□ MG)        DD            ⇐ !,  meta_ssolve MG DD.
meta_wsolve  (mg G GG)     DD            ⇐ !,  ssolve G GG DD.
```

Figure 8.9: Generalizing meta-interpreter: Meta-Goal solution.

becomes bound to **obj1**, while that of *DD* remains uninstantiated.

As both boxed and unboxed clauses are used in the proofs we have developed, the reader might rightfully expect both to appear in *DD*, the resulting sufficient conditions of the generalization. In fact, this is the case: for the goal □ **a** ⇒ **b** ⇒ (**a** , **b**), our generalizing meta-interpreter produces the explanation-based generalization (**a** , *G*) ⇐ □ **a**, *G*. However, boxed clauses are 'necessarily' true, and hence need not be re-checked during the application of a derived rule. Instead, it is the conjunction of the utilized unboxed clauses which constitutes the simplest expression of the sufficient conditions for *GG*. Removing boxed clauses from *DD* requires a simple reduction predicate, whose definition may be found in Appendix A.4. The result of simplifying the above (**a** , *G*) ⇐ *G*. We take this approach as it is easier to remove boxed clauses from the completed generalization than to avoid their initial incorporation, since only top-level boxed clauses could reasonably be recognized in **solve**. (For the meta-interpreter implementation, these simplification predicates will unavoidably destroy degenerate generalizations such as the above — *i.e.*, ones with variables at the top-level.)

8.7 Operationality

Incorporating operationality criteria within the preceding prototype requires providing the meta-interpreter with access to an operationality predicate **oper**. The revision involves inserting the following clause at the head of the **solve** predicates. We illustrate the change for **wsolve**; an analogous change is necessary in **ssolve**:

```
wsolve  G  GG  DD  ⇐  oper G, !,
                     DD = GG, wsolve_orig G.
```

where **wsolve_orig** is the version of **wsolve** that does not perform EBG — *i.e.*, that given within Figure 8.5. The computation proceeds in the same manner, but EBG is suspended during the solution of operational subgoals. Instead, *DD* is bound to the current generalized

goal *GG*, which, because it is operational, becomes one of the sufficient conditions of the resulting generalization. The above clause is expected to be used for recursive invocations of **solve** — *i.e.*, during the solution of subgoals; if a top-level goal is made operational, the resulting explanation-based generalization is trivial.

It is the user's responsibility to specify the computation necessary to determine **oper** of particular goals. Should no clauses be provided for **oper**, the above implementation behaves in the same manner as the original. And since the definition of **oper** may be extended in the course of the current computation, this formulation permits dynamic operationality criteria (although it does not support the expression of preconditions for dynamic operationality criteria; see §4.5).

8.8 Assimilation

We demonstrated within Chapter 5 that **rule** cannot be implemented within λProlog. On the other hand, the generalizing interpreter of §8.6 illustrates that λProlog is sufficient for the realization of EBG itself. The question is, then, whether this meta-interpreter can be extended so that explanation-based generalizations may be learned (*i.e.*, assimilated)? The answer is no, and the reason, as the reader might expect, is λProlog's inability to universally generalize: consider that the tentative implementation of **lemma_ebg**

$$\textbf{lemma\_ebg}\ \ G\ \ K\ \ \Leftarrow\ \ \textbf{wsolve}\ G\ GG\ DD,\ ((GG \Leftarrow DD)\ \Rightarrow\ K).$$

typically allows the assimilated generalization to be applied only once (*i.e.*, for one instantiation of its variables).

It is pleasing that **lemma_ebg** *can* be implemented in terms of **rule** and **solve**:

$$\textbf{lemma\_ebg}\ \ G\ \ K\ \ \Leftarrow\ \ \textbf{rule}\ \ (\textbf{wsolve}\ G\ GG\ DD)\ (\forall G\ \forall GG\ \forall DD.\ \textbf{wsolve}\ G\ GG\ DD\ \Rightarrow\ (GG \Leftarrow DD))\ K.$$

Similarly, consider the following encoding of **rule_ebg**:

$$\textbf{rule\_ebg}\ \ G\ \ (\Box\ F)\ \ K\ \ \Leftarrow\ \ \textbf{rule}\ \ (\textbf{wsolve}\ G\ GG\ DD,\ \textbf{instan}\ F\ GG\ E)\ (\forall G\ \forall GG\ \forall DD\ \forall E\ \forall F.\ (\textbf{wsolve}\ G\ GG\ DD,\ \textbf{instan}\ F\ GG\ E)\ \Rightarrow\ (E \Leftarrow DD))\ K.$$

where

$$\textbf{instan}\ \ (\forall\ F)\ \ \ G\ \ D\ \ \Leftarrow\ \ \textbf{instan}\ \ (F\ x)\ G\ D.$$
$$\textbf{instan}\ \ (G \Rightarrow D)\ \ G\ \ D.$$

The **instan** predicate simply replaces universally quantified variables with new logical variables, and then does the appropriate unification between the generalized goal and the goal associated with the forward inference step.

8.9 The Barcan Formula

The Barcan formula, for a higher-order logic, is as follows:

$$(\forall x.\, \Box\, (Px)) \;\Rightarrow\; (\Box\, \forall x.\, Px)$$

While the converse of Barcan — that is,

$$(\Box\, \forall x.\, Px) \;\Rightarrow\; (\forall x.\, \Box\, (Px))$$

is true in all modal logics, the validity of the Barcan formula varies. Under $\lambda^{\Box}$Prolog's inference system, the Barcan formula is indeed valid.[2,3]

Although in terms of the logic there is no difference between the left- and right-hand sides of Barcan, the generalization algorithm does distinguish the two. That is, the relative order of $\Box$ and $\forall$, while not affecting provability, *can* affect EBG! In particular, variables whose universal quantifiers are outside of $\Box$ are not abstracted within the generalized proof. To illustrate, we once again employ the suicide example of §4.2: If the clause

$$!!\ \textbf{weapon}\ z \Leftarrow \textbf{gun}\ z.$$

which is simply shorthand for

$$\Box\, \forall z.\, \textbf{weapon}\ z \Leftarrow \textbf{gun}\ z.$$

were replaced instead with

$$\forall z.\, \Box\, (\textbf{weapon}\ z \Leftarrow \textbf{gun}\ z).$$

the resulting generalization becomes

$$!!\ \textbf{kill}\ x\ x \Leftarrow \textbf{depressed}\ x,\ \textbf{buy}\ x\ \textbf{obj1},\ \textbf{gun obj1}.$$

The reader's initial impression may be that the above violates the partition established between domain and training theory, since **obj1** only appears in the latter, but yet makes its way into the derived rule. Observe, however, that the universal quantification $\forall z$ occurs

---

[2]It is also the case that *S5* includes Barcan.

[3]Disallowing the Barcan formula would significantly complicate the inference system. Consider that the ordering of $\forall$ and $\Box$ is presently irrelevant. An alternative inference system not admitting Barcan would have to maintain the additional context of whether or not universal variables occurred within the scope of a $\Box$.

Similarly, the manner in which universal variables are treated in λProlog would complicate disallowing Barcan. The problem is that quantifiers are not maintained during computation, but rather are replaced by special place-holding *uvars*, for 'universal variables.' This means that the problem of maintaining scope inside or outside of $\Box$ would require two distinct kinds of uvars: one for x, and the other for y within $\forall x.\, \Box\, \forall y.\, D$.

outside of the $\Box$, and thus the clause incorporated within the generalized proof does not include that quantification. Instead, the utilized clause may be viewed as

$$\Box\,(\textbf{weapon}\ z \Leftarrow \textbf{gun}\ z).$$

Because the universal quantification occurs within the training theory, we chose not to universally generalize z within the domain theory. This decision really just provides an additional level of expressiveness: for the large majority of situations, users will presumably want the original interpretation, which is easily achieved with the !! notation.

As an aside, if we were to instead require that EBG not discriminate between the left- and right-hand sides of Barcan, we would thereby avoid the need for the !! notation: consider that

$$!!\ \textbf{weapon}\ z \Leftarrow \textbf{gun}\ z.$$

is equivalent to

$$\Box\,\forall z.\ \textbf{weapon}\ z \Leftarrow \textbf{gun}\ z.$$

which is equivalent (under the Barcan formula) to

$$\forall z.\ \Box\,(\textbf{weapon}\ z \Leftarrow \textbf{gun}\ z).$$

which, in turn, can be expressed as simply

$$\Box\,(\textbf{weapon}\ z \Leftarrow \textbf{gun}\ z).$$

by relying upon λProlog's implicit universal quantification of top-level clauses. It is unclear, however, whether this alternative EBG algorithm could be realized through revisions to the generalizing meta-interpreter of §8.6.

Chapter 9

$\lambda^{\Box}_{\mathcal{G}}$Prolog Implementation

In §3.4.3 we made mention of eLP, the implementation of λProlog written in COMMON LISP and developed at Carnegie Mellon University by Conal Elliott and Frank Pfenning in the framework of the Ergo project [38]. Then within §8.5 and 8.6, we introduced interpreters written in λProlog for the modal logic $\lambda^{\Box}$Prolog and for that logic extended with explanation-based generalization.

These prototype implementations of $\lambda^{\Box}$Prolog have been extremely valuable for experimenting with different variations of both the logic and the EBG algorithm, and moreover for providing a formal specification of each. They are, however, extremely slow due to the additional level of interpretation, which also precludes the application of lower-level implementation strategies (such as hashing rules based upon predicate names). Such optimizations are not directly expressible within λProlog (that is, not without substantially complicating the encoding). Furthermore, these meta-interpreters are not sufficiently powerful to handle λProlog primitives (*e.g.*, cut and arithmetic), or to realize the !! convention, or to implement our primitives for initiating and controlling generalization and assimilation, **rule** and **rule_ebg** (Chapter 5).

We have addressed these deficiencies by extending our existing λProlog interpreter, eLP, with □, !!, **rule**, and **rule_ebg**. The nature of these extensions is the topic of this chapter.

9.1 Implementing □

The first addition we made to eLP was the modal logic operator □. The necessary extensions to the eLP interpreter largely follow the abstract $\lambda^{\Box}$Prolog interpreter developed in §8.5: goals are subject to two levels of solution — strong (for boxed goals) and weak (for unboxed), and similarly, boxed clauses and subclauses applied in the course of a proof are distinguished from their unboxed counterparts. Rather than supplying two pairs of COMMON LISP routines analogous to **wsolve** & **ssolve** (which reduce complex goals to atomic ones), and **wmatch** & **smatch** (which use the logic program to derive atomic goals), we simply included a boolean context argument within the corresponding interpreter routines.

λProlog clause normal-form. eLP employs the normal-form representation of λProlog clauses described in §3.6: recall that D_{nf} may be defined as

$$\begin{array}{rcl} D_{\mathsf{nf}} & ::= & D_{\forall} \mid D_{\mathsf{nf}}\,,\, D_{\mathsf{nf}} \\ D_{\forall} & ::= & D_{\Leftarrow} \mid \forall x.\, D_{\forall} \\ D_{\Leftarrow} & ::= & A \Leftarrow G \end{array}$$

In this way, $\mathcal{P}$ is mapped to a set of clauses such that each D may be represented as an atomic clause-head A, a goal precondition G (which implies A), and a list of universal variables $\mathcal{X}$ over which D is universally quantified. The motivation for using D_{nf} is two-fold: (1) clauses may thereby be simply represented as three components — A, G, and $\mathcal{X}$, and (2) the relevance of a given clause to a particular atomic goal G_{a} may be easily determined by unifying G_{a} and A.

Once general G-forms have been reduced to atomic ones, the eLP interpreter operationally proceeds as follows:

- Select a set of clauses from $\mathcal{P}$ which may be applicable to the solution of G_{a}. This step includes insuring that A and G_{a} begin with the same predicate, and potentially makes use of further matching optimizations such as indexing (whereby subterms of G_{a} are matched against pre-selected subterms of A).
- Create new logical (or existential) variables $\hat{\mathcal{X}}$ for each universal variable in $\mathcal{X}$, and then substitute $\hat{\mathcal{X}}$ for $\mathcal{X}$ in A and G, yielding $\hat{A}$ and $\hat{G}$.
- Unify $\hat{A}$ and G_{a}. If unsuccessful, backtrack and chose another clause.
- Recursively solve $\hat{G}$.

λ$^{\Box}$Prolog clause normal-form. For the same reasons as within eLP, we desire to make use of a normal-form for λ$^{\Box}$Prolog clauses within □eLP. Recall that within §8.2, we developed the following λ$^{\Box}$Prolog normal-form:

$$\begin{array}{rcl} D_{\mathsf{nf}} & ::= & D_{\mathsf{nf}}\,,\, D_{\mathsf{nf}} \mid D_{\forall} \\ D_{\forall} & ::= & \forall x.\, D_{\forall} \mid A \Leftarrow G^{\mathsf{w}} \mid (\Box\, D_{\Box\forall}) \Leftarrow G^{\mathsf{w}} \\ D_{\Box\forall} & ::= & \forall y.\, D_{\Box\forall} \mid A \Leftarrow G^{\mathsf{s}} \end{array}$$

λ$^{\Box}$Prolog clauses may thereby be represented either in the preceding λProlog normal-form, or else as an atomic clause-head A, a weak enabling goal G^{w}, a strong enabling goal G^{s} (either of which may be **true** in the degenerate case), a set of universal variables $\mathcal{X}$ that appears outside the scope of the optional □, and a set of universal variables $\mathcal{Y}$ that appears within the scope of □.

Rather than actually maintaining two normal-forms, □eLP uses the extended normal-form with the additional inclusion of a boolean flag indicative of weather the original clause contained □. This distinction is necessary in order to differentiate $(\Box\, A) \Leftarrow G$ and $A \Leftarrow G$, as the former is sufficient for deriving $\Box\, A$, while the latter is not.

As within eLP, $\lambda^{\Box}$Prolog goals are reduced to atomic subgoals for solution, as the meta-interpreter of §8.6 illustrates. Once goals have been so reduced, the operational behavior of the □eLP interpreter may be characterized as follows (as summarized in §8.3):

- Select a series of clauses from $\mathcal{P}$ which may be applicable to the solution of $G_{\mathbf{a}}$. If $G_{\mathbf{a}}$ is being strongly solved, we may ignore clauses of the unboxed normal-form; *i.e.*, those that do not contain □.
- Create new logical variables $\hat{\mathcal{W}}$ and $\hat{\mathcal{S}}$ for each universal variable in $\mathcal{W}$ and $\mathcal{S}$, and then substitute $\hat{\mathcal{W}}$ for $\mathcal{W}$ and $\hat{\mathcal{S}}$ for $\mathcal{S}$ in A, $G^{\mathbf{w}}$, and $G^{\mathbf{s}}$, yielding $\hat{A}$, $\hat{G}^{\mathbf{w}}$, and $\hat{G}^{\mathbf{s}}$, respectively.
- Unify $\hat{A}$ and $G_{\mathbf{a}}$. If unsuccessful, backtrack and chose another clause.
- For strong solution, solve $(\hat{G}^{\mathbf{w}}\ ,\ \Box\ \hat{G}^{\mathbf{s}})$. For weak solution, solve $(\hat{G}^{\mathbf{w}}\ ,\ \hat{G}^{\mathbf{s}})$.

The '!!' notation. In order to ease programming within our $\lambda^{\Box}_{\mathcal{C}}$Prolog prototype, we included the !! convention (introduced in §4.3) for top-level $\lambda^{\Box}$Prolog clauses. The realization of !! is particularly straightforward, as it simply requires merging $\mathcal{W}$ into $\mathcal{S}$ within the above representation.

9.2 Implementing "rule"

In Chapter 5 we established that because there is no provision for universally quantifying existing free variables, **rule** is not implementable within a λProlog meta-interpreter. Thus for us to actually experiment with the construct and to run the examples we have presented, it was necessary to implement **rule** within eLP.

Recall from §5.3.2 that the alternative operational definition of

$$\mathcal{P} \vdash \mathbf{rule}\ G\ (\forall \mathcal{X}.\, G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ K$$

is as follows:

1. Create new logical variables $\hat{\mathcal{X}}$ for each universal variable in $\mathcal{X}$, and then substitute $\hat{\mathcal{X}}$ for $\mathcal{X}$ in $G_{\mathcal{X}}$ and $D_{\mathcal{X}}$, yielding $G_{\hat{\mathcal{X}}}$ and $D_{\hat{\mathcal{X}}}$.
2. Unify G and $G_{\hat{\mathcal{X}}}$.
3. Solve G.
4. Let $\mathcal{Y} = \mathsf{free}(G) - \mathsf{free}(\mathcal{P})$
5. Solve $(\forall \hat{\mathcal{X}}.\forall \mathcal{Y}.\, D_{\hat{\mathcal{X}}}) \Rightarrow K$.

$\forall\hat{\mathcal{X}}$ is correct as those variables of $\hat{\mathcal{X}}$ which **rule** has instantiated no longer appear free in $D_{\hat{\mathcal{X}}}$ (a side-effect of unification).

The implementation of **rule** within this framework is straightforward, save for two considerations:

- Correctly limiting $\mathcal{Y}$ so that that the derived assumption is not *over-general* in that it universally generalizes variables free within pending assumptions.
- Handling the heretofore unaddressed topic of higher-order constraints.

9.2.1 Variables Free in Assumptions

We emphasized in Chapter 5 that the universal generalization step associated with **rule** must not quantify variables free in assumptions (*i.e.*, within $\mathcal{P}$), as such a generalization violates the declarative nature of the **rule** construct. The most straightforward solution would be simply for □eLP to maintain a set $\mathcal{F}$ of all the variables currently free in $\mathcal{P}$. This would require that before making the local assumption D, that $\mathcal{F}$ first be augmented with any variables free in D. (Subsequent instantiation of variables within $\mathcal{F}$ does not pose a problem, as instantiated variables no longer occur in goals and clauses.) **rule**'s universal generalization step, then, involves subtracting this set from the candidates for universal generalization.

A problem with the above strategy is that it is potentially computationally expensive. A second alternative is to maintain the set of local assumptions (that potentially contain free variables). To determine which variables to universally generalize, these assumptions are searched for occurrences of these candidates. While a brute-force approach to this search would be even more expensive than the preceding algorithm, the use of time stamps on variables and expressions should substantially reduce this overhead: through time stamps, older expressions need not be searched for occurrences of newer variables.[1,2]

Within the existing system, neither of the above strategies is implemented. Instead, within our prototype we made the expedient choice of universally generalizing *all* free variables within an assumption! This has not proven to be a problem for experimentation, but is would certainly be unacceptable for anything further.

9.2.2 Constraints

In §5.3.4 we mentioned that higher-order unification requires the accumulation of constraints. Simply put, such constraints are necessary to represent unifications that do not result in

[1]For time stamping to be effective, it is necessary that unification cannot result in the instantiation of old variables with newer ones, as this would require re-stamping the containing expressions. Instead, unification is realized by binding newer variables to old.

[2]Didier Ramy of INRIA has shown that time stamping leads to substantially improved performance for the related problem of closing type variables within the ML **let** construct.

variable instantiation: recall from §3.2 that $f\mathbf{a} = g\mathbf{aa}$ allows the variable f to be instantiated with any of $\lambda x.g\mathbf{aa}$, $\lambda x.gx\mathbf{a}$, $\lambda x.g\mathbf{a}x$, or $\lambda x.gxx$, none of which is an instance of another. Similarly g could be instantiated with $\lambda x.\lambda y.f\mathbf{a}$, $\lambda x.\lambda y.fx$, or $\lambda x.\lambda y.fy$. By representing $f\mathbf{a} = g\mathbf{aa}$ as a unification constraint, subsequent computation is free to instantiate f or g with any of the above.

Since these constraints are essential to the process of higher-order unification, they must be incorporated within the assumptions derived through **rule**. Within eLP, constraints are passed along as part of the interpretation environment. This is why, for example, we did not need to treat them within our meta-interpreters of Chapter 8: that detail of $\lambda^{\Box}$Prolog implementation was handled directly within λProlog. However, the universal generalization of variables disqualifies the existing $\Box$eLP constraints. Thus we need a means by which to capture the persistent constraints required by **rule**.

Within the D-forms assumed by **rule** in $\Box$eLP, constraints are represented simply as a conjunction of higher-order (unification) equations over λProlog terms. Hence, as mentioned, the actual form for **rule**'s assumption is $\forall\mathcal{X}'\forall\mathcal{Y}.\,\sigma_{\mathcal{X}}\theta D_{\mathcal{X}} \Leftarrow \exists\mathcal{Z}.\,C_{\mathcal{Z}}$, where $C_{\mathcal{Z}}$ is the conjunction of the pending constraints, and $\mathcal{Z}$ represents all variables occurring only in $C_{\mathcal{Z}}$. We return to this topic in §9.3.2.

9.3 Implementing "rule_ebg"

Recall from §5.4, the eLP-oriented operational interpretation of

$$\textbf{rule\_ebg}\ G\ (\Box\,\forall\mathcal{X}.\,G_{\mathcal{X}} \Rightarrow D_{\mathcal{X}})\ K$$

is as follows:

1. Solve G with EBG enabled, resulting in the explanation-based generalization

$$\Box\,\forall\mathcal{Y}.\,\theta GG_{\mathcal{Y}} \Leftarrow \theta DD_{\mathcal{Y}}.$$

2. Create new logical variables $\hat{\mathcal{X}}$ for each universal variable in $\mathcal{X}$, and then substitute $\hat{\mathcal{X}}$ for $\mathcal{X}$ in $G_{\mathcal{X}}$ and $D_{\mathcal{X}}$, yielding $G_{\hat{\mathcal{X}}}$ and $D_{\hat{\mathcal{X}}}$.

3. Create new logical variables $\hat{\mathcal{Y}}$ for each universal variable in $\mathcal{Y}$, and then substitute $\hat{\mathcal{Y}}$ for $\mathcal{Y}$ in $GG_{\mathcal{Y}}$ and $DD_{\mathcal{Y}}$, yielding $GG_{\hat{\mathcal{Y}}}$ and $DD_{\hat{\mathcal{Y}}}$.

4. Unify $GG_{\hat{\mathcal{Y}}}$ and $G_{\hat{\mathcal{X}}}$.

5. Solve $(\Box\,\forall\hat{\mathcal{Y}}.\,\forall\hat{\mathcal{X}}.\,D_{\hat{\mathcal{X}}} \Leftarrow DD_{\hat{\mathcal{Y}}}) \Rightarrow K.)$

```
wsolve  !                 !                  true  ⇐ !.
wsolve  (once G)          (once GG)          DD    ⇐ !,  once (wsolve G GG DD).
wsolve  nl                nl                 true  ⇐ !,  nl.
wsolve  (writesans S)     (writesans SS)     true  ⇐ !,  writesans S.
wsolve  (write M)         (write MM)         true  ⇐ !,  write M.
wsolve  (read G)          (read GG)          DD    ⇐ !,  read λt. ∃tt.
                                                           wsolve (G t) (GG tt) (DD tt).
```

Figure 9.1: Generalizing meta-interpreter: Special goals.

9.3.1 Specials

We have defined explanation-based generalization over computation expressed in the logic of $\lambda^{\square}$Prolog, but we have not discussed how a generalizing interpreter could handle extra-logical features, such as !, input/output, or arithmetic. Cut and input/output are especially relevant as they have been used to define the interactive problem solvers introduced in Chapters 6 and 7. Yet, these extra-logical constructs do not appear in the explanation-based generalizations we have illustrated thus far. This is because the programs that include these features, for example the **interactive** tactical of §6.1, are boxed, and therefore do not occur in the resulting derived rules.[3]

Implementing generalization over these constructs is problematic in that they can not be realized at the level of our abstract interpreter. Essentially, specials are handled by executing them for the particular case (*i.e.*, the current goal), and incorporating an analog within the associated generalization. Figure 9.1 illustrates this strategy in the treatment of several pertinent λProlog specials within the generalizing meta-interpreter of §8.6.

9.3.2 Higher-order Constraints and EBG

Just as for **rule**, the higher-order constraints associated with an explanation-based generalization must be represented in the *D*-form assumped by **rule_ebg**, and as you would expect, this is handled in an identical fashion.

Complexity of higher-order constraints. However, the constraint sets that result from EBG tend to be substantially more complicated than those associated with **rule**. Appendix A.7 gives a listing of the most complicated set of constraints we have yet encountered: that resulting from the program transformation scenario of Chapter 7. Even to those well-versed in λProlog, these constraints are inscrutable.

For more complex higher-order generalizations to be truly useful, methods must be devised for making higher-order constraints more palatable to the programmer. There remains the

[3]Nevertheless, the EBG algorithm as we have defined it initially includes specials that occur within boxed clauses in *DD*, its accumulator for EBG preconditions (§8.6). These are then removed in the course of the simplification that eliminates necessarily true (boxed) preconditions

possibility (discussed briefly in Chapters 3 & 10) that higher-order unification (and hence EBG), as it is presently defined, is too general to be of value for more complex situations. Perhaps a more restricted unification algorithm could effectively capture the generalization without yielding an inscrutable result. In other words, higher-order unification may itself be such a rich mechanism that explanation-based generalizations based upon it are potentially over-general. For now, the question remains open.

Performance. There is a more practical concern following from the complexity of these higher-order constraints, and that is performance. Somehow our explicit representation of higher-order constraints as λ-term unification equations can lead to substantially increased computation costs, again for our more complex examples. In fact, for the program transformation illustration of Chapter 7, the application of the explanation-based generalization takes longer than its generation! Obviously this is grossly unacceptable for anything more than a experimental system, since it violates one of the primary missions of EBG — improving performance.

There are a couple of contributing factors to this performance degradation. One expects that the computation involved in applying an explanation-based generalization to a goal should be a subset of that required to solve the goal without generalization. Given this, the observed performance loss must demonstrate inappropriate representation choices for the derived generalizations. From this, it is easy to target the complex constraint equations associated with the derived rules in question. We believe that the additional overhead incurred through our explicit representation of constraints as λ-terms is at least partially responsible. In particular, this inefficiency may be the result of information lost in the copying of terms (for example, once identical ("**eq**") terms becoming merely equivalent). If this is correct, effective performance will require better data structures for maintaining constraints across $\lambda^{\Box}$Prolog computations.

The above is particularly problematic when combined with a limitation of eLP's present higher-order unification algorithm — it can behave *eagerly* in its commitment to particular solutions [39]. When the unification algorithm makes the wrong guess, significant amounts of backtracking is required. For the larger constraint sets, this cost may be substantial. Instead, what one would like is maximally lazy unification.

While the above speculation is not satisfying, we believe that the exploration of higher-order constraint representation and satisfaction is itself a substantial research problem. Thus far, our efforts have focused on defining the language and learning mechanisms of $\lambda^{\Box}_{\mathcal{C}}$Prolog. We anticipate that the further study of higher-order constraints and unification will lead to scrutable encodings and fast algorithms. Indeed, the long term relevance of this work is dependent upon success in this area.

Chapter 10

Conclusion

10.1 Summary

As stated at the outset, this thesis should broadly be viewed as a language design effort. The result, $\lambda^{\Box}_{G}$Prolog, encompasses extensions to λProlog that afford rudimentary modal reasoning, higher-order EBG, and logically-motivated constructs for controlling generalization and assimilation. By incorporating learning mechanisms within the logic programming language, $\lambda^{\Box}_{G}$Prolog defers to the programmer the problem of determining when to learn. It is our belief that only the system designer is generally positioned to effectively address this problem, leveraging his familiarity with both the problem domain and the problem solver. Thus, while $\lambda^{\Box}_{G}$Prolog is not itself a learning system, it is intended to serve as a high-level foundation for the implementation of such systems.

Through the framework of $\lambda^{\Box}_{G}$Prolog, this thesis offers a number of contributions:

- The use of the modal operator $\Box$ to provide an alternative formulation of EBG.
- The extension of the EBG algorithm to treat a higher-order representation language, and then formulation of higher-order EBG via a $\lambda^{\Box}$Prolog meta-interpreter.
- A logically-sound mechanism, **rule**, for universal generalization within logic programs.
- A generalization of the **rule** construct, **rule_ebg**, to afford EBG.
- The integration of all of the above in an environment that supports user-guided problem solving and learning.
- A prototype implementation $\Box$eLP.
- A suite of examples.

Just as this thesis borrows from a number of different areas, we believe its results are relevant to a range of research efforts: formal methods for higher-order domains (*e.g.*, program development, theorem proving, and natural language); λProlog and logic programming; explanation-based generalization and machine learning; modal logic; and language design in general.

10.2 Future Work

10.2.1 Further Experimentation with $\lambda^{\Box}$Prolog

While we have presented a number of examples in the course of developing this thesis, the majority of our efforts have been devoted to constructing the $\lambda^{\Box}_{\mathcal{G}}$Prolog framework rather than exploring its application. Thus, one of the primary future directions is further experimentation. λProlog is currently being used as a research vehicle in a number of different areas: logics, programming languages, and natural language [106, 88, 85, 103]. It would be interesting to further consider what impact $\lambda^{\Box}_{\mathcal{G}}$Prolog can have on these domains, particularly since other researchers have already done the lion's share of the necessary formalization by programming in λProlog. A particularly attractive domain to which we have as of yet given only cursory consideration is the proofs-as-programs paradigm introduced in §2.1. Like the other domains considered herein, proofs-as-programs is interesting to consider because of the relevance of higher-order expressivity and the reliance upon interactive proof development.

In another direction, we have done relatively little in the way of re-implementing examples from the EBG literature. This, of course, is because we have been primarily focused upon the higher-order domains that initially motivated our work. However, another worthwhile means of exercising $\lambda^{\Box}_{\mathcal{G}}$Prolog would be to consider the encoding of more extensive first-order problems. Of particular interest is the further consideration of the possible interplay between the $\Box$ operator and dynamic operationality criteria.

Of the many remaining questions, perhaps the predominant one is whether a relatively complete, higher-order apprentice learning system can be effectively realized within $\lambda^{\Box}_{\mathcal{G}}$Prolog (or within any other language, for that matter). While we have provided example scenarios in Chapters 6 and 7, we have not yet produced such a system. This is in part due to limitations within our present implementation, both in terms of performance (discussed in Chapter 9), and in terms of functionality, such as the lack of a mouse interface (discussed in Chapters 7). This is also, of course, due to time constraints.

10.2.2 Practical Considerations

In Chapter 9 we raised a number of limitations that presently hinder $\lambda^{\Box}_{\mathcal{G}}$Prolog application to more complex domains:

- Higher-level support for interaction
- Higher-order unification, constraints, and lazy unification

Another issue is that of instrumentation for performance measurement and analysis: while we have deferred to the programmer the task of determining when EBG is invoked and how its results are exploited, $\lambda^{\Box}_{\mathcal{G}}$Prolog does not provide the designer with the facilities required to guide this process and ensure that performance actually improves. While we have not investigated the issue, we believe that such tools could be smoothly integrated with $\lambda^{\Box}_{\mathcal{G}}$Prolog.

10.2.3 Future Work on λProlog and Explanation-Based Generalization

λProlog and EBG are fundamentally melded within this dissertation, yet each continues to evolve through current research:

For λProlog, among other efforts, Miller is presently considering the restriction of λProlog to a subset L_λ that admits decidable unification and most general unifiers. It is unclear to what extent explanation-based generalizations of L_λ computation are themselves legal L_λ clauses. There remains the possibility, however, that L_λ's restrictions upon higher-order unification would address the problems associated with complex higher-order constraints (see §9.3.2).

For EBG, in addition to work we have already cited, researchers are presently extending the paradigm to generalize from failure as well as success [25], and to more effectively generalize over iterative and recursive computations [19, 118]. (To illustrate the inadequacy of $\lambda^{\Box}_{\mathcal{G}}$Prolog in this latter respect, consider from Chapter 6 that the explanation-based generalization of the **repeat** tactical commits to the particular number of iterations used within the example solution.)

Even more speculatively, and of particular interest to us, is the further development of the 'language-based' approach to learning (of which $\lambda^{\Box}_{\mathcal{G}}$Prolog is an exemplar) to encompass other EBG methodologies, other paradigms of generalization (*e.g.*, similarity-based methods [65, 12]), and analogical problem solving techniques [10, 15, 23, 57, 98].

10.2.4 Logical Foundations of □ and EBG

While we have defined an interpreter for $\lambda^{\Box}$Prolog, we have not considered the underlying modal semantics of our intuitionistic calculus. What, for example, is its relationship to S4 and S5 [16, 69]? Also, the higher-order EBG algorithm we have illustrated should be verified. Finally, it is worth further considering the formal relationship between □, EBG, and operationality criteria.

10.2.5 Incorporating "rule" and "rule_ebg" within other Logic Programming Languages

On an alternative front, one could consider the incorporation of **rule** (and also **rule_ebg**) within Prolog as a declarative alternative to **assert** and **retract**. Assuming the necessary syntax to express variable binding (for the second argument of **rule**), **rule** could be directly added to Prolog. Presumably, this would be accomplished without the addition of implication and the explicit scope K, but instead by extending the program in the manner of Prolog's **lemma** (introduced within §5.2.1). The implementation would be substantially easier than in λProlog, since Prolog programs are always closed, and thus we can simply quantify over all logic variables (in the manner of **assert**) without having to check for variables free in current assumptions. The implementation problems posed by **rule** in the context of Prolog are thus very similar to those associated with **assert**, and variations on the techniques proposed by Lindholm & O'Keefe [79] are applicable.

10.2.6 Logical Foundations of "rule" and "rule_ebg"

Intuitively, logic programming proofs require the repeated solution of instances of common goals. More formally, the execution of a logic program usually produces what is known as a *cut-free* or *normal* proof of the query (actually, they belong to the even more restrictive class of *uniform proofs* [86]). This is no longer true for a language extended with **rule** and **rule_ebg**, wherein derived proofs may be substantially shorter than non-normal proofs (which is the reason why **rule** and **rule_ebg** are effective). Can we characterize the class of deductions which can be found by programs employing **rule**? By those using **rule_ebg**? Is there a way to extend these constructs within the logic programming paradigm so that even more general deductions can be found?

Finin, *et al.* have considered a more complete integration of forward and backward chaining [46]. Their approach supports extended computations in both directions by allowing the programmer to write both forward and backward chaining Horn clauses. We do not see, however, a way to express the interplay between forward and backward reasoning required by **rule** within their language. It would be interesting to consider whether their approach could be fruitfully combined with the higher-order constructs and scoping available in λProlog, and then also with **rule** and **rule_ebg**.

10.2.7 Ramifications of "rule"

Dale Miller has pointed out that **rule** permits the formulation of at least the extra-logical predicate **flexible** (**flexible** is the higher-order equivalent of **var**): **flexible** M indicates that M's head is a variable — that is, that M can be unified with any term. He provides the definition

$$\textbf{flexible}\ M \Leftarrow \forall p, q.\ (\forall x.\, p\, x) \Rightarrow \textbf{rule}\ (p\, M)\ (\forall x.\, p\, x \Rightarrow q\, x)\ (\forall x.\, q\, x)$$

In some ways, **flexible**'s extra-logical nature makes $\lambda^{\square}_{G}$Prolog less declarative:

$$?-\ \textbf{flexible}\ X,\ X = \mathbf{a}.$$

succeeds, while

$$?-\ X = \mathbf{a},\ \textbf{flexible}\ X$$

does not.

10.2.8 Alternatives to "rule" and "rule_ebg"

There are a few examples closely related to the ones we have given herein for which **rule** and **rule_ebg** do not appear to be powerful enough. The problem is that it is not possible to translate the universal quantifiers introduced during the universal generalization step into explicit quantifiers at the object-level. That is, there is no means by which the universally quantified assumption can be accessed by the program. Finding a declaratively and operationally satisfactory solution is yet another topic for future research.

Appendix A

Some Unabridged Examples

Within this Appendix we include the unabridged λProlog source code that produced many of the examples upon which this dissertation has so heavily relied.

Notation. Up to this point, I have made a substantial effort at attractively typeset λProlog syntax. However, this does not seem to be a worthwhile endeavor for the more extended examples to follow. Thus, it becomes necessary to introduce the ASCII-restricted syntax of eLP — our implementation of λProlog described in Chapter 9.

Instead of italics and boldface, eLP λ-term variables begin with an uppercase character, while λ-term constants are lowercase. This same distinction is make for type constants and type variables. In fact, the only exception is that variables explicitly bound by λ may be of either case (since constants may never follow a λ).

λ-abstraction is represented with \, which binds the variable preceding it; that is, \ acts as an infix λ, such as within `x:A\y:B\(f x y)`, which is equivalent to $\lambda x{:}A.\lambda y{:}B.\mathbf{f}xy$. (This is probably the most difficult aspect of eLP syntax to get used to.)

The following table summarizes the mapping from TEX to ASCII:

λProlog	eLP
`c`	`c`
x	`X`
$\longrightarrow$	`->`
$\lambda x.$	`x\`
$\forall x.$	`pi x\`
$\exists x.$	`sigma x\`
$\Box$	`box`
$\Rightarrow$	`=>`
$\Leftarrow$	`<=`
	`:-`
x^a	`expn x a`
x/a	`x div a`

A.1 Clause simplification

```
module simplify.

type    false           o.
type    simpl           o -> o -> o.
type    simpl1          o -> o -> o.

simpl   true        true    :- !.
simpl   (H1 , H2)   H       :- !, simpl H1 H1i, simpl H2 H2i,
                               simpl1 (H1i , H2i) H.
simpl   (H1 ; H2)   H       :- !, simpl H1 H1i, simpl H2 H2i,
                               simpl1 (H1i ; H2i) H.
simpl   (H1 => H2)  H       :- !, simpl H1 H1i, simpl H2 H2i,
                               simpl1 (H1i => H2i) H.
simpl   (pi H1)     H       :- !, pi x\(simpl (H1 x) (H1i x)),
                               simpl1 (pi H1i) H.
simpl   (sigma H1)  H       :- !, pi x\(simpl (H1 x) (H1i x)),
                               simpl1 (sigma H1i) H.
simpl   (box H1)    H       :- !, simpl H1 H1i,
                               simpl1 (box H1i) H.
simpl   Ha          Ha      :- !.


simpl1  true            true      :- !.

simpl1  (true , H2)     H2        :- !.
simpl1  (H1 , true)     H1        :- !.
simpl1  (false , H2)    false     :- !.
simpl1  (H1 , false)    false     :- !.

simpl1  (true ; H2)     true      :- !.
simpl1  (H1 ; true)     true      :- !.
simpl1  (false ; H2)    H2        :- !.
simpl1  (H1 ; false)    H1        :- !.

simpl1  (true => H2)    H2        :- !.
simpl1  (H1 => true)    true      :- !.
simpl1  (false => H2)   true      :- !.
simpl1  (H1 => false)   false     :- !.

simpl1  (pi X\ true)    true      :- !.
simpl1  (sigma X\ true) true      :- !.
simpl1  (pi X\ false)   false     :- !.
simpl1  (sigma X\ false) false    :- !.

simpl1  (box true)      true      :- !.
simpl1  (box false)     false     :- !.

simpl1  H               H         :- !.
```

A.2 Rudimentary Resolution Theorem Prover

This implementation incorporates a *unit strategy* — that is, use smallest clauses first. It relies upon user interaction to control the whole process.

```
module resolve_unit.
import simplify.

type clause_            o -> int -> o.
type min_clause         o -> int -> o.
type min_clause_aux     o -> int -> int -> o.
type resolve            o -> o -> o -> o.
type rtp                o.
type rtp_aux            o -> int -> o -> o.
type write_clauses      o.


resolve (P ; Q)  S         (P ; R)  :- resolve Q S R.
resolve (P ; Q)  S         (Q ; R)  :- resolve P S R.
resolve S        (P ; Q)   (P ; R)  :- resolve Q S R.
resolve S        (P ; Q)   (Q ; R)  :- resolve P S R.

resolve P        (not P)   false.
resolve (not P)  P         false.


min_clause P P# :- min_clause_aux P P# 1.

min_clause_aux P P# Size :-
  (clause_ P Size, P# = Size) ;
  (Size < 10,  Size1 is Size + 1,  min_clause_aux P P# Size1).


rtp :- rule (rtp_aux R R# G)
            (pi R\ (pi R#\ (clause_ R R# :- rtp_aux R R# G)))
            G.


rtp_aux R R# G :-
  min_clause P P#,
  min_clause Q Q#,

  resolve P Q R1, simpl R1 R,
  R# is ((P# + Q#) - 2),
  not (clause_ R R#),          %% This is a hack, one may be an instance of the
                               %% other.
  nl,
  nl, writesans "First clause  : ", write P,
  nl, writesans "Second clause : ", write Q,
  nl, writesans "Resolved to   : ", write R,
```

```
  nl, writesans "Assume resolvant? [y,r,q,t] ",
  read ans\ ( (ans = n, fail) ;                    %% no cut here, we want to
                                                   %% backtrack
              (ans = q, !, G = write_clauses) ;
              (ans = t, !, G = top)
              (ans = y, !, G = rtp)).


write_clauses :-
  nl, nl, writesans "Resolution complete",
  nl, nl, writesans "Clauses: ", !,
  ((min_clause R R#, nl, writesans "  ", write R, fail);nl).
```

```
eval true       true          :- !.
eval (G1 , G2) (G3 , G4)      :- !, eval G1 G3,
                                   eval G2 G4.
eval (G1 ; G2) G              :- !, (eval G1 G; eval G2 G).
eval (D => G)  G1             :- !, ndform D D1, hyp D1 => eval G G1.
eval (pi G)    (pi G1)        :- !, pi X\ (eval (G X) (G1 X)).
eval (sigma G) (sigma G1)     :- !, eval (G X) (G1 X).


eval Gatom     SubG           :- hyp D,
                                  select D (Gatom :- SubG1),
                                  eval SubG1 SubG.
eval Gatom     SubG           :- Gatom, SubG = true, !.
eval Gatom     Gatom.
```

Figure A.1: Evaluator.

A.3 Partial evaluator for λProlog

The section contains a more thorough development of the partial evaluator **peval** originally presented within §4.6. We also provide a more extensive application of **peval**: the partial evaluation of a interpreter with respect to a particular object language.

A.3.1 A λProlog Evaluator

To simplify the presentation of the partial evaluator, we first introduce a λProlog evaluator given in Figure A.1. To a large degree this evaluator follows both the original presentation of **peval** and the λProlog meta-interpreter (§8.5). We include it nevertheless for the sake of completeness.

The predicate **eval** reduces one goal (its first argument) to another (its second).[1] In the last three clauses of **eval**, an atomic goal is reduced either (1) by applying a relevant clause from the object-logic program $\mathcal{P}_{ob}$ (enumerated via **hyp**), (2) by evaluating it directly in λProlog, or (3) by a no-op. For (1) the **select** predicate, the code for which may be found in Figure A.2, nondeterministically selects a candidate clause to reduce the goal G_a. **select** replaces universal variables with new logic variables to facilitate the unification of the clause head and G_a. For (2) we avoid coding the evaluation of special goals (*e.g.*, !, =, or arithmetic) by realizing them directly (*i.e.*, reflecting them) within λProlog. (To be practical, the above evaluator should also simplify the result of evaluation before yielding an answer.)

[1] While we could accurately use the term 'meta-evaluator' (since both the object- and meta-language are λProlog), it would become cumbersome when we turn to discussion of the 'partial-meta-evaluator.'

```
select     Ds          D :- select_and Ds D.

select_and (D1 , D2) D :- !, (select_and D1 D;
                              select_and D2 D).
select_and D1           D :- select_pis D1 D.

select_pis (pi D1)   D :- !, select_pis (D1 X) D.
select_pis D         D.
```

Figure A.2: Clause selection.

A.3.2 A Partial Evaluator

The partial evaluator of Figure A.3 is identical to the evaluator, except that rather than directly applying clauses (1) or directly interpreting goals in λProlog (2), **peval** *queries* the user before committing to any such operation; that is, **peval** is a user-guided partial evaluator. Those queries are handled by the auxiliary predicates given in Figure A.4.

This brings us back to the primary distinction between PE and EBG: partial evaluation requires substantial amounts of search control in order to produce interesting specializations. The partial evaluator includes no notion of $\mathcal{D}$ and $\mathcal{T}$, but the same results may be achieved by explicitly *not* partially evaluating goals reduced by specific clauses.[2] Deriving all the rules that can be produced through PE is equivalent to finding the deductive closure of a logic program. (Of course, heuristics can potentially reduce this problem by guiding PE toward more interesting specializations.) EBG, on the other hand, uses an example solution (as well as □ and operationality criteria) to determine what combination of clauses will in essence be partially evaluated.

If the partially evaluated logic program is again to be interpreted (*e.g.*, through **eval** & **peval**), the following top-level is sufficient: (Recall that all object clauses are accessed through the predicate **hyp**.)

```
peval_top E (E :- G) :-
   rule (peval E G)
        (pi E\ (pi G\ (peval E G => hyp (E :- G))))
        top.
```

A.3.3 An Example Application

As a more extended example of partial evaluation and reflection, consider the meta-interpreter of Figure A.5, which is taken from Takeuchi & Furukawa [127]. (The concept is borrowed from Shapiro [117].) This meta-interpreter combines uncertainty or confidence factors with its solution of goals. We shall apply this meta-interpreter to the object-program

[2] Actually, this is not quite the case, since PE is like operationality criteria in that it does permit internal proof steps to be abstracted from the generalized proof (§4.5) as does our formulation of EBG.

```
peval true      true        :- !.
peval (G1 , G2) (G3 , G4)   :- !, peval G1 G3,
                               peval G2 G4.
peval (G1 ; G2) (G3 ; G4)   :- !, peval G1 G3,
                               peval G2 G4.
peval (D => G)  G1          :- !, ndform D D1, hyp D1 => peval G G1.
peval (pi G)    (pi G1)     :- !, pi X\ (peval (G X) (G1 X)).
peval (sigma G) (sigma G1)  :- !, peval (G X) (G1 X).

peval Gatom     SubG        :- hyp D,
                               select D (Gatom :- SubG1),
                               do_rule Gatom (Gatom :- SubG1) SubG.
peval Gatom     SubG        :- do_bottom Gatom SubG.
peval Gatom     Gatom.
```

Figure A.3: Evaluator.

```
do_rule Gatom (Gatom :- SubG1) SubG :- nl,
      nl, writesans "Goal to be partial evaluated: ", nl,
      writesans " ", write Gatom, nl,
      nl, writesans "Selected rule: ", nl,
      writesans " ", write Gatom, writesans " :- ", nl,
      writesans "   ", write SubG1, nl,
      nl, writesans "Use this rule? [y|n|s|e] ",
      read ans\
         ((ans = s, !, SubG = SubG1) ;
          (ans = n, !, fail) ;
          (ans = y, !, peval SubG1 SubG) ;
          (ans = e, !, eval SubG1 SubG) ;
          (!, nl, writesans "Illegal command: ", write ans, nl,
           do_rule Gatom (Gatom :- SubG1) SubG)).

do_bottom G SubG :- nl,
      nl, writesans "Goal to be partial evaluated: ", nl,
      writesans " ", write G, nl,
      writesans "No more rules apply.", nl,
      writesans "Evaluate it directly in eLP (e.g., 'is','<')? [y|n] ",
      read ans\
         ((ans = n, !, SubG = G) ;
          (ans = y, !, G, SubG = true) ;
          (!, nl, writesans "Illegal command: ", write ans, nl,
           do_bottom G SubG)).
```

Figure A.4: Partial Evaluator.

```
module meta_medic.
import lists.

type    msolve          o -> list int -> o.
type    mrule           o -> o -> int -> o.
type    cft             int -> list int -> int -> o.
type    product         list int -> int -> int -> o.
type    cf_rule         o -> int -> o.

msolve true    (100 :: nil).
msolve (A , B) Z            :- msolve A X, msolve B Y, append X Y Z.
msolve (not A) (Cf :: nil)  :- msolve A (C :: nil), C < 20, Cf is 100 - C.
msolve A       (Cf :: nil)  :- mrule A B F, msolve B S, cft F S Cf.

cft X Y Z :- product Y 100 Y1, Z is ((X * Y1) div 100).

product nil A A.
product (X :: L) A X1 :- B is ((X * A) div 100), product L B X1.

mrule A B    F :- cf_rule (A :- B) F.
mrule A true F :- cf_rule A F.
```

Figure A.5: Meta-program with certainty factors.

---

of Figure A.6, which is concerned with the prescription of drugs (and is also taken from Takeuchi & Furukawa [127]). For example, **eval** permits the solution of the following goal

```
( hyp (cf_rule (suffers_from scott peptic_ulcer :- true)  0 :- true),
  hyp (cf_rule (complains_of scott pain :- true) 100 :- true)
) => eval (msolve (should_take scott aspirin) Cf) G
```

yielding `G = true` and `Cf = (42 :: nil)`.

This meta-interpreter may be partially evaluated with respect to this object program, certain results of which are illustrated within Figures A.7 & A.8. (Each of these derived clauses is reported in Takeuchi & Furukawa [127]. We duplicate their results here because the examples are informative, and more importantly, that the results illustrate an application of **rule**.)

It is essential to understand that we are herein dealing with three levels of language: **peval**, the meta-interpreter **msolve**, and the medical object language. One problem with always interpreting the results of partial evaluation is simply the extra cost incurred through this extensive layering of language.[3] As we have discussed, the solution is to reflect the program being manipulated (in this case, the combination of meta- and object-program) into the logic program, and thereby run it directly. This may be accomplished via a previously listed revision to our top-level:

---

[3]This factor depends upon the nature of the interpreter and object language, but between one and two orders of magnitude appears to be typical for λProlog.

```
module medic.
import meta_medic.

kind person             type.
kind drug               type.
kind symptom            type.
kind condition          type.

type should_take        person -> drug -> o.
type complains_of       person -> symptom -> o.
type suppresses         drug -> symptom -> o.
type unsuitable         drug -> person -> o.
type aggravates         drug -> condition -> o.
type suffers_from       person -> condition -> o.

cf_rule (should_take Person Drug :- complains_of Person Symptom,
                                    suppresses Drug Symptom,
                                    not (unsuitable Drug Person))
        70.
cf_rule (suppresses aspirin pain)
        60.
cf_rule (suppresses lomotil diarrhoea)
        65.
cf_rule (unsuitable Drug Person :- aggravates Drug Condition,
                                   suffers_from Person Condition)
        80.
cf_rule (aggravates aspirin peptic_ulcer)
        70.
cf_rule (aggravates lomotil impaired_liver_function)
        70.
```

Figure A.6: Object-program.

```
msolve (should_take Y1 Y2) (Y :: nil) :-
    msolve (complains_of Y1 Y3) Y6,
    msolve (suppresses Y2 Y3) Y9,
    msolve (unsuitable Y2 Y1) (Y11 :: nil),
    Y11 < 20, Y10 is 100 - Y11,
    append Y9 (Y10 :: nil) Y7, append Y6 Y7 Y4,
    cft 70 Y4 Y.
msolve (suppresses aspirin pain) (60 :: nil).
msolve (suppresses lomotil diarrhoea) (65 :: nil).
msolve (unsuitable Y10 Y12) (Y4 :: nil) :-
    msolve (aggravates Y10 Y11) Y15,
    msolve (suffers_from Y12 Y11) Y17,
    append Y15 Y17 Y13, cft 80 Y13 Y4.
msolve (aggravates aspirin peptic_ulcer) (70 :: nil).
msolve (aggravates lomotil impaired_liver_function) (70 :: nil).
```

Figure A.7: Results of partial evaluation.

```
msolve (should_take Y7 aspirin) (Y4 :: nil) :-
    msolve (complains_of Y7 pain) Y12,
    msolve (suffers_from Y7 peptic_ulcer) Y27,
    cft 80 (70 :: Y27) Y18,
    Y18 < 20, Y17 is 100 - Y18,
    append Y12 (60 :: Y17 :: nil) Y10,
    cft 70 Y10 Y4.
msolve (should_take Y7 lomotil) (Y :: nil) :-
    msolve (complains_of Y7 diarrhoea) Y6,
    msolve (suffers_from Y7 impaired_liver_function) M,
    cft 80 (70 :: M) Y15),
    Y15 < 20,  Y14 is 100 - Y15,
    append Y6 (65 :: Y14 :: nil) Y4,
    cft 70 Y4 Y.
```

Figure A.8: Further results of partial evaluation.

```
peval_top E (E :- G) :-
   rule (peval E G)
        (pi E\ (pi G\ (peval E G => (E :- G))))
        top.
```

A.4 Generalizing interpreter for $\lambda^{\Box}$Prolog

For the sake of completeness, we list the unabridged λProlog implementation discussed in §8.5 & §8.6.

```
module metaebg.

type  gsolve o -> o -> o.

type  hyp o -> o.

type  wsolve o -> o -> o -> o.
type  ssolve o -> o -> o -> o.

type  wmatch o -> o -> o -> o -> o -> o.
type  smatch o -> o -> o -> o -> o -> o.
type  bmatch o -> o -> o -> o -> o -> o.

type  meta_wsolve o -> o -> o.
type  meta_ssolve o -> o -> o.

type  breduce o -> o -> o.
type  reduce o -> o -> o.
type  reduce1 o -> o -> o.

type  dosolve o -> o -> o -> o.
```

```
wsolve true        true          true          :- !.
wsolve (G1 , G2)   (GG1 , GG2)   (DD1 , DD2)   :- !, wsolve G1 GG1 DD1,
                                                    wsolve G2 GG2 DD2.
wsolve (G1 ; G2)   (GG1 ; GG2)   DD            :- !, (wsolve G1 GG1 DD;
                                                     wsolve G2 GG2 DD).
wsolve (D => G)    GG            DD            :- !, hyp D => wsolve G GG DD.
wsolve (pi G)      (pi GG)       (DD X)        :- !, pi X\
                                                     (wsolve (G X) (GG X) (DD X)).
wsolve (sigma G)   (sigma GG)    (DD T)        :- !, wsolve (G T) (GG T) (DD T).
wsolve (box G)     (box GG)      DD            :- !, ssolve G GG DD.
wsolve Ga          GGa           DD1           :-    ghyp D DD,
                                                    wmatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.
wsolve Ga          GGa           (DD , DD1)    :- !, hyp D,
                                                    wmatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.


ssolve true        true          true          :- !.
ssolve (G1 , G2)   (GG1 , GG2)   (DD1 , DD2)   :- !, ssolve G1 GG1 DD1,
                                                    ssolve G2 GG2 DD2.
ssolve (pi G)      (pi GG)       (DD X)        :- !, pi X\
                                                     (ssolve (G X) (GG X) (DD X)).
ssolve (box G)     (box GG)      DD            :- !, ssolve G GG DD.
ssolve Ga          GGa           DD1           :-    ghyp D DD,
                                                    smatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.
ssolve Ga          GGa           (DD , DD1)    :- !, hyp D,
                                                    smatch D Ga DD GGa MG,
                                                    meta_wsolve MG DD1.

ssolve (G1 ; G2)   (GG1 ; GG2)   DD
       :- !, error (writesans "Illegal disjunction in boxed goal").
ssolve (D => G)    (DD1 => GG)   DD2
       :- !, error (writesans "Illegal implication in boxed goal").
ssolve (sigma G)   (sigma GG)    (DD T)
       :- !, error (writesans "Illegal existential in boxed goal").
```

```
wmatch (D1 , D2)  Ga  DD           GGa  MG :- !, (wmatch D1 Ga DD GGa MG;
                                                  wmatch D2 Ga DD GGa MG).
wmatch (G => D)   Ga  (GG => DD)   GGa  (gsolve G GG, MG)
                                           :- !, wmatch D Ga DD GGa MG.
wmatch (pi D)     Ga  DD           GGa  MG :- !, wmatch (D X) Ga DD GGa MG.
wmatch (box D)    Ga  (box D)      GGa  MG :- !, bmatch D Ga D GGa MG.
wmatch Ga         Ga  GGa          GGa  true.



smatch (D1 , D2)  Ga  DD           GGa  MG :- !, (smatch D1 Ga DD GGa MG;
                                                  smatch D2 Ga DD GGa MG).
smatch (G => D)   Ga  (GG => DD)   GGa  (gsolve G GG, MG)
                                           :- !, smatch D Ga DD GGa MG.
smatch (pi D)     Ga  DD           GGa  MG :- !, smatch (D X) Ga DD GGa MG.
smatch (box D)    Ga  (box D)      GGa  (box MG)
                                           :- !, bmatch D Ga D GGa MG.



bmatch (D1 , D2)  Ga  (DD1 , DD2)  GGa  MG :- !, (bmatch D1 Ga DD1 GGa MG;
                                                  bmatch D2 Ga DD2 GGa MG).
bmatch (G => D)   Ga  (GG => DD)   GGa  (gsolve G GG, MG)
                                           :- !, bmatch D Ga DD GGa MG.
bmatch (pi D)     Ga  (pi DD)      GGa  MG :- !, bmatch (D  X:A) Ga
                                                        (DD Y:A) GGa MG.
bmatch (box D)    Ga  (box D)      GGa  MG :- !, bmatch D Ga D GGa MG.
bmatch Ga         Ga  GGa          GGa  true.



wmatch (D1 ; D2)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal disjunction in program").
wmatch (sigma D)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal existential in program").


smatch (D1 ; D2)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal disjunction in program").
smatch (sigma D)  Ga  DD       GGa  MG
    :- !, error (writesans "Illegal existential in program").


bmatch (D1 ; D2)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal disjunction in program").
bmatch (sigma D)  Ga  DD  GGa  MG
    :- !, error (writesans "Illegal existential in program").
```

```
   % Meta-interpreter invocation.

dosolve  G GG DD :- !, wsolve G GG DD1, breduce DD1 DD2, reduce DD2 DD.
```

```
    % Solution of accumulated Meta-goals.

meta_wsolve true           true         :- !.
meta_wsolve (MG1 , MG2)    (DD1 , DD2)  :- !, meta_wsolve MG1 DD1,
                                              meta_wsolve MG2 DD2.
meta_wsolve (box MG)       DD           :- !, meta_ssolve MG DD.
meta_wsolve (gsolve G GG)  DD           :- !, wsolve G GG DD.



meta_ssolve true           true         :- !.
meta_ssolve (MG1 , MG2)    (DD1 , DD2)  :- !, meta_ssolve MG1 DD1,
                                              meta_ssolve MG2 DD2.
meta_ssolve (box MG)       DD           :- !, meta_ssolve MG DD.
meta_ssolve (gsolve G GG)  DD           :- !, ssolve G GG DD.
```

```
    % Replaces "(box H)" with "true" in DD --- the set of sufficient
    % conditions.

breduce  true        true          :- !.
    % Above should be first to avoid infinite recursion on logical variables.
    % This allows uninstaniated variables to be 'reduced' out of the picture.
    % (Good for all but degenerate higher-order generalizations.)
breduce  (H1 , H2)   (H1i , H2i)   :- !, breduce H1 H1i, breduce H2 H2i.
breduce  (H1 ; H2)   (H1i ; H2i)   :- !, breduce H1 H1i, breduce H2 H2i.
breduce  (H1 => H2)  (H1i => H2i)  :- !, breduce H1 H1i, breduce H2 H2i.
breduce  (pi H)      (pi Hi)       :- !, pi X\ (breduce (H X) (Hi X)).
breduce  (sigma H)   (sigma Hi)    :- !, pi X\ (breduce (H X) (Hi X)).
breduce  (box H)     true          :- !.
breduce  Ha          Ha            :- !.



    % Simplifies sufficient conditions by removing superfluous true's.

reduce   true         true         :- !.
    % Above should be first to avoid infinite recursion on logical variables.
    % This allows uninstaniated variables to be 'reduced' out of the picture.
    % (Good for all but degenerate higher-order generalizations.)

reduce   (H1 , H2)   H             :- !, reduce H1 H1i, reduce H2 H2i,
                                         reduce1 (H1i , H2i) H.
reduce   (H1 ; H2)   H             :- !, reduce H1 H1i, reduce H2 H2i,
                                         reduce1 (H1i ; H2i) H.
reduce   (H1 => H2)  H             :- !, reduce H1 H1i, reduce H2 H2i,
                                         reduce1 (H1i => H2i) H.
reduce   (pi H1)     H             :- !, pi X\ (reduce (H1 X) (H1i X)),
                                         reduce1 (pi H1i) H.
reduce   (sigma H1)  H             :- !, pi X\ (reduce (H1 X) (H1i X)),
                                         reduce1 (sigma H1i) H.
reduce   (box H1)    H             :- !, reduce H1 H1i,
                                         reduce1 (box H1i) H.
reduce   Ha          Ha            :- !.


reduce1   true             true    :- !.
reduce1   (true , H2)      H2      :- !.
reduce1   (H1 , true)      H1      :- !.
reduce1   (true ; H2)      true    :- !.
reduce1   (H1 ; true)      true    :- !.
reduce1   (true => H2)     H2      :- !.
reduce1   (H1 => true)     true    :- !.
reduce1   (pi X\ true)     true    :- !.
reduce1   (sigma X\ true)  true    :- !.
reduce1   (box true)       true    :- !.
reduce1   H                H       :- !.
```

A.5 Tactic-style Integration

A.5.1 Tactics for Integration

```
module integrate_tac.
import tacticals.

type    expn            int -> int -> int.
type    -               int -> int.
type    log             int -> int.
type    cos             int -> int.
type    sin             int -> int.

type    intgr           (int -> int) -> (int -> int) -> o.

type    dx              name.
type    cf              name.
type    p1              name.
type    pm1             name.
type    pw              name.
type    cfl             name.
type    cfr             name.
type    pl              name.
type    cos_            name.


!! tac dx  (intgr  x\1                 x\x)
           true.

!! tac cf  (intgr  x\A                 x\(A * x))
           true.

!! tac p1  (intgr  x\x                 x\((expn x 2) div 2))
           true.

!! tac pm1 (intgr  x\(expn x (- 1))    x\(log x))
           true.

!! tac pw  (intgr  x\(expn x A)        x\((expn x (A + 1)) div (A + 1)))
           true.

!! tac cfl (intgr  x\(A * (B x))       x\(A * (Bi x)))
           (intgr  x\(B x)            x\(Bi x)).

!! tac cfr (intgr  x\((B x) * A)       x\((Bi x) * A))
           (intgr  x\(B x)            x\(Bi x)).

!! tac pl  (intgr  x\((A x) + (B x))  x\((Ai x) + (Bi x)))
           ((intgr x\(A x)            x\(Ai x)) ,
            (intgr x\(B x)            x\(Bi x))).
```

```
tac cos_   (intgr      cos    sin)
           true.
```

A.5.2 Tacticals

```
module tacticals.
import simplify.

kind    name            type.

type    tac             name -> o -> o -> o.
index   tac             1.

type    maptac          name -> name.
type    then            name -> name -> name.
type    orelse          name -> name -> name.
type    repeat          name -> name.
type    idtac           name.
type    try             name -> name.
type    complete        name -> name.
type    quit            name.
type    stop            name -> name.
type    interact_solve  name.


!! tac (maptac T) true true.

!! tac (maptac T) (OGa , OGb) OG :- !,
     tac (maptac T) OGa OGa1,
     tac (maptac T) OGb OGb1,
     simpl (OGa1 , OGb1) OG.

!! tac (maptac T) (OGa ; OGb) OG :- !,
     tac (maptac T) OGa OGa1,
     tac (maptac T) OGb OGb1,
     simpl (OGa1 ; OGb1) OG.

!! tac (maptac T) IG OG :-
     tac T IG OG1,
     simpl OG1 OG.
```

```
!! stop_tac ok.
!! stop_tac pop_.


!! tac top_solve (pi OG1)     OG    :- !, pi X\(tac top_solve (OG1 X) (OG2 X)),
                                       simpl (pi OG2) OG.

!! tac top_solve (OG1 , OG2) OG    :- !, tac top_solve OG1 OG3,
                                       tac top_solve OG2 OG4,
                                       simpl (OG3 , OG4) OG.

!! tac top_solve OG1          OG    :- !, tac interact_solve OG1 OG.



!! tac interact_solve true true :- !,
      nl, writesans "--- Subtree solved ...", nl, nl.

!! tac interact_solve OG1  OG    :- !,
      nl, writesans "Goal to be reduced: ", nl, nl,
      write OG1, nl,
      nl, writesans "Enter rule ",
      read T\(tac T OG1 OG2, simpl OG2 OG3,
             ((stop_tac T, OG = OG3);
              (tac top_solve OG3 OG)).


!! tac ok              OG  OG   :- !.
!! tac pop_            OG  OG   :- !.
!! tac top_            OG  OG   :- !, top.



!! tac (ebg_ Name) OG1 OG    :- !,
       rule_ebg (tac top_solve OG1 OG2)
                (pi OGa\ (pi OGb\
                  (tac Name OGa OGb :- tac top_solve OGa OGb)))
                (nl, nl, writesans "--- New rule is a generalization of",
                 nl, nl, write (tac Name OG1 OG2), nl, nl,
                 ((simpl OG2 true, !, top) ; tac top_solve OG2 OG)).




!! tac direct    (pi OG1)     OG    :- !, pi X\(tac direct (OG1 X) (OG2 X)),
                                       simpl (pi OG2) OG.

!! tac direct    (OG1 , OG2) OG    :- !, tac direct OG1 OG3,
                                       tac direct OG2 OG4,
                                       simpl (OG3 , OG4) OG.
```

```
!! tac direct    OG1          OG    :- tac Name OG1 OG2, simpl OG2 OG3, !,
                                       ( (OG3 = true, !, OG = true)
                                        ;(tac direct OG3 OG, !)
                                        ;OG3 = OG
                                       ).
```

```
module tail_rec_itac.
import tactical_iebg.

 % Transformational program development mapping recursive applicative
 % functions (of a certain form) to tail-recursive.  Illustrates
 % higher-order EBG.
 % This variation is designed to work with interactive generalizing
 % tactics.

 % Scott Dietzen, 1990

kind   exp                      type.

type   ife                      exp -> exp -> exp -> exp.
type   not_                     exp -> exp.
type   lam                      (exp -> exp) -> exp.
type   appl                     exp -> exp -> exp.
type   fix                      (exp -> exp) -> exp.
type   equals                   exp -> exp -> exp.
type   times                    exp -> exp -> exp.
type   minus                    exp -> exp -> exp.
type   nil_                     exp.
type   cons_                    exp -> exp -> exp.
type   null                     exp -> exp.
type   car                      exp -> exp.
type   cdr                      exp -> exp.
type   append                   exp -> exp -> exp.
type   zero                     exp.
type   one                      exp.

type   associative              (exp -> exp -> exp) -> o.
type   commutative              (exp -> exp -> exp) -> o.
type   left_identity            (exp -> exp -> exp) -> exp -> o.
type   right_identity           (exp -> exp -> exp) -> exp -> o.



type   getrev          exp -> o.
type   getdiff         exp -> o.
type   getfact         exp -> o.


type   insert_lam      (exp -> exp) -> name.
type   add_oper_rid    (exp -> exp -> exp) ->
                            ((exp -> exp) -> exp) -> name.
type   abstract_arg    (exp -> exp) -> (exp -> exp) -> name.
type   name_fn         exp -> (exp -> exp) -> name.
type   unfold          (exp -> exp) -> name.
type   reduce_1        ((exp -> exp) -> exp) -> name.
type   dist_ife_2      (exp -> exp -> exp) ->
                            ((exp -> exp -> exp) -> exp) -> name.
type   left_id_2       (exp -> exp -> exp) ->
```

```
                          ((exp -> exp -> exp) -> exp) -> name.
type   assoc_2          (exp -> exp -> exp) ->
                          ((exp -> exp -> exp) -> exp) -> name.
type   fold_two_3       ((exp -> exp -> exp -> exp) -> exp) ->
                          (exp -> exp -> exp -> exp) ->
                          exp -> name.


right_identity minus zero.

associative times.
commutative times.
left_identity  times one.
right_identity times one.

associative    append.
left_identity  append nil_.
right_identity append nil_.



getfact  (fix Fact\ (lam N\
           (ife (equals N zero) one (times (appl Fact (minus N one)) N)))).

getrev  (fix Rev\ (lam L\
          (ife (null L) nil_
                        (append (appl Rev (cdr L)) (cons_ (car L) nil_))))).

getdiff (fix Diff\ (lam L\
          (ife (null L) zero (minus (car L) (appl Diff (cdr L)))))).
```

```
!! tac (insert_lam C)
    (C (fix f\(lam n\(G f n))))
    (C (lam m\(appl (fix f\(lam n\(G f n))) m))).


!! tac (add_oper_rid Op C)
    (C x\(G x))
    (C x\(Op (G x) A))                          :- right_identity Op A.


!! tac (abstract_arg C1 C2)
    (C1 (C2 A))
    (C1 (appl (lam m\(C2 m)) A)).


!! tac (name_fn Fnew C)
    (C G)
    (C (fix fnew\G))                            :- Fnew = (fix fnew\G).


!! tac (unfold C)
    (C (fix f\(G f)))
    (C (G (fix f\(G f)))).


!! tac (reduce_1 C)
    (C x\(appl (lam n\(G n)) x))
    (C x\(G x)).


!! tac (dist_ife_2 Op C)
    (C x\y\(Op (ife (Bool x y) (E1 x y) (E2 x y)) (H x y)))
    (C x\y\(ife (Bool x y) (Op (E1 x y) (H x y))
                           (Op (E2 x y) (H x y)))).


!! tac (left_id_2 Op C)
    (C x\y\(Op A (H x y)))
    (C x\y\(H x y))                                 :- left_identity Op A.


!! tac (assoc_2 Op C)
    (C x\y\(Op (Op (H1 x y) (H2 x y)) (H3 x y)))
    (C x\y\(Op (H1 x y) (Op (H2 x y) (H3 x y))))    :- associative Op.


!! tac (fold_two_3 C1 C2 (fix f\(lam m\(lam n\(C2 F n m))))) % no occur f
    (C1 f\x\y\(C2 F (H1 x y) (H2 x y)))                    % no occur f
    (C1 f\x\y\(appl (appl f (H2 x y)) (H1 x y))).

        %% C1 -- context within input program to be replaced.
        %% C2 -- matches some function F in both the original definition
```

```
%% and the input program
```

```
%%%%%%%%%% The invocation:

% getfact F, tac top_solve F Fout.


%%%%%%%%%% The inputs as prompted

% 1. insert_lam          G\G.

% 2. add_oper_rid times G\(lam n\(G n)).

% 3. abstract_arg        G\G
                         G\(lam n\(times (W0 n) G)).

% 4. name_fn        Fnew G\(appl G W).

% 5. unfold              G\(appl (fix f1\(lam m\(lam n\
                                    (times (appl G n) m)))) W).

% 6. reduce_1            G\(appl (fix f1\(lam m\(lam n\
                                    (times (G n) m)))) W).

% 7. dist_ife_2   times G\(appl (fix f1\(lam m\(lam n\(G m n)))) W).

% 8. left_id_2    times G\(appl (fix f1\(lam m\(lam n\
                              (ife (W1 m n) (G m n) (W3 m n))))) W).

% 9. assoc_2      times G\(appl (fix f1\(lam m\(lam n\
                              (ife (W1 m n) (W2 m n) (G m n))))) W).

%10. fold_two_3          G\(appl (fix f1\(lam m\(lam n\
                                 (ife (W1 m n) (W2 m n) (G f1 m n))))) W)
                           G\H1\H2\(times (appl G H1) H2)
                           Fnew.
%11. ok.
```

A.7 Constraints

We illustrate the nature of the higher-order constraints with the set of constraint equations coming out the program transformation example introduced in Chapter 7.

```
[
    <Op (appl G1 (F101 m f1)) (F10 m f1) == F9 m f1>
    <Op (appl G1 n) h2 == F3 h2 n>
    <F9 x x1 == Op (H1 x x1) (Op (H2 x x1) (H3 x x1))>
    <Op (Op (H1 m1 f11) (H2 m1 f11)) (H3 m1 f11) == F71 m1 f11>
    <F6 x2 x3 == Op (G2 x3) x2>
    <Op (appl (lam N \ (G2 N)) x4) x5 == F5 x5 x4>
    <Op (lam N \ (G3 m2 N)) (H m2) ==
       Op (lam N1 \ (appl (fix F \ (lam N \ (G F N))) N1)) m2>
    <F3 n1 m3 == Op (G3 n1 m3) (H n1)>
    <Op (appl (fix F \ (G4 F)) n2) m4 == F3 m4 n2>
    <F5 x6 x7 == Op (appl (G4 (fix F \ (G4 F))) x7) x6>
    <Op (ife (F7 m5 f12) (E1 m5 f12) (E2 m5 f12)) (H4 m5 f12) == F6 m5 f12>
    <Op DD (F8 m6 f13) == Op (E1 m6 f13) (H4 m6 f13)>
    <F71 m7 f14 == Op (E2 m7 f14) (H4 m7 f14)>
]
Prefix Fragment:
    [sigma DD F10 F101 F3 F5 F6 F7 F71 F8 F9 G Op][sigma G3 H](pi m2 n1 m3)
    [sigma G4](pi m4 n2 x6 x7)[sigma G2](pi x5 x4 x2 x3)[sigma E1 E2 H4]
    (pi m5 f12 m7 f14 m6 f13)[sigma H1 H2 H3](pi m1 f11 x x1)[sigma G1]
    (pi h2 n m f1)
```

Appendix B

Bibliography

Bibliography

[1] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, September 1983.

[2] Robert Balzer, Thomas E. Cheatham Jr., and Cordell Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, pages 38–45, November 1983.

[3] Jon Barwise. Mathematical proofs of computer system correctness. *Notices of the American Mathematical Society*, pages 844–851, September 1989.

[4] CIP System Group: F. L. Bauer et al. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[5] F. L. Bauer. From specification to machine code: Program construction through formal reasoning. In *Sixth International Conference on Software Engineering*, 1982.

[6] Jon Bentley. *Writing Efficient Programs*. Prentice–Hall, 1982.

[7] Jon Bentley. *More Programming Pearls: Confessions of a Coder*. Addison–Wesley, 1988.

[8] Neeraj Bhatnagar. A correctness proof of explanation-based generalization as resolution theorem proving. In *Proceedings of the AAAI Spring Symposium on Explanation-Based Learning*, pages 220–225, 1988.

[9] Anthony J. Bonner, L. Thorne McCarty, and Kumar Vadaparty. Expressing database queries with intuitionistic logic. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 831–850, Cambridge, Massachusetts, 1989. MIT Press.

[10] Bishop Brock, Shaun Cooper, and William Pierce. Some experiments with analogy in proof discovery. Technical Report AI–347–86, MCC, Austin, TX, October 1986. Preliminary Version.

[11] Manfred Broy and Peter Pepper. Program development as a formal activity. *IEEE Transactions on Software Engineering*, SE-7(1):44–67, 1977.

[12] Wray Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36:149–176, 1988.

[13] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[14] A.D. Burt, P.M. Hill, and J.W. Lloyd. Preliminary report on the logic programming language Gödel. Technical Report TR–90–02, Department of Computer Science, University of Bristol, March 1990.

[15] Jaime G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning II*, pages 371–392. Morgan Kaufmann, 1986.

[16] Brian F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1980.

[17] Widong Chen, Michael Kifer, and David S. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference, 1989, Volume 2*, Cambridge, Massachusetts, 1989. MIT Press.

[18] E.J. Chikofsky, editor. *Computer-Aided Software Engineering.* Software Development Series. IEEE, 1989.

[19] W.W. Cohen. Generalizing number and learning from multiple examples in explanation-based learning. In *Proceedings of the Fifth International Machine Learning Conference*, June 1988.

[20] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[21] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[22] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[23] Thierry Boy de la Tour and Ricardo Caferra. Proof analogy in interactive theorem proving: A method to express and use it via second order pattern matching. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 95–99, Seattle, WA, July 1987.

[24] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[25] Gerald DeJong and Raymond Mooney. Explanation-based generalization: An alternate view. *Machine Learning*, 1(2):145–176, 1986.

[26] Luis Fariñas del Cerro. Molog: A system that extends Prolog with modal logic. *New Generation Computing*, 4:35–50, 1986.

[27] Luis Fariñas del Cerro and Martti Penttonen. A note on the complexity of the satisfiability of modal Horn Clauses. *Journal of Logic Programming*, 4(1):1–10, 1987.

[28] Nachum Dershowitz. Program abstraction and instantiation. *TOPLAS*, 7(3):446–477, 1985.

[29] T. M. Dietterich. Learning at the knowledge level. *Machine Learning*, 1(3):287–315, 1986.

[30] T. M. Dietterich et al. Learning and inductive inference. In Paul R. Cohen and Edward A. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 3, pages 325–511. William Kaufmann, 1982.

[31] Scott Dietzen and Frank Pfenning. A declarative alternative to assert in logic programming. In Vijay Saraswat and Kazunori Ueda, editors, *International Logic Programming Symposium*, pages 372–386. MIT Press, October 1991.

[32] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 1991. To appear.

[33] Scott Dietzen and William L. Scherlis. Analogy in program development. In J. C. Boudreaux, B. W. Hamill, and R. Jernigan, editors, *The Role of Language in Problem Solving 2*, pages 95–117. North-Holland, 1987. Also available as Ergo Report 86–013, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[34] E. W. Dijkstra. *A Disipline of Programming.* Prentice-Hall, Inc., 1976.

[35] Edsger W. Dijkstra. On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12):1397–1414, December 1989. (The above pages also include responses. Discussion continues in the forum of CACM: March 1990 and April 1990.).

[36] Edsger W. Dijkstra, editor. *Formal Development of Programs and Proofs.* Addison-Wesley, 1990.

[37] Michael R. Donat and Lincoln A. Wallen. Learning and applying generalised solutions using higher order resolution. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 41–60, Berlin, May 1988. Springer-Verlag LNCS 310.

[38] Conal Elliott and Frank Pfenning. eLP: A Common Lisp implementation of λProlog in the Ergo Support System. Available via ftp over the Internet, October 1989. Send mail to elp-request@cs.cmu.edu on the Internet for further information.

[39] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[40] Thomas Ellman. Explanation-based learning: A survey of programs and perspectives. *ACM Computing Surveys*, 21(2):163–221, June 1989.

[41] Oren Etzioni. *A structural theory of explanation-based learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1990. Available as CMU-CS-90-185.

[42] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In *IFIP TC2 Working Conference on Program Specification and Transformation.* North-Holland, 1986.

[43] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989.

[44] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 61–80, Berlin, May 1988. Springer-Verlag LNCS 310.

[45] James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

[46] Tim Finin, Rich Fritzson, and Dave Matuszek. Adding forward chaining and truth maintenance to Prolog. In *IEEE Conference on Artificial Intelligence Applications*, March 1989. Also available as Paoli Research Center technical report PRC–LBS–8802.

[47] D. M. Gabbay and U. Reyle. N-prolog: an extension of Prolog with hypothetical implications I. *Journal of Logic Programming*, 1(4):319–355, 1985.

[48] Susan L. Gerhart. Applications of formal methods: Development of virtuoso software. *IEEE Software*, 7(5):6–10, September 1990. (Special issue on formal methods).

[49] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[50] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[51] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[52] J.V. Guttag and J.J. Horning. Formal specification as a design tool. In *Seventh Symposium on the Principles of Programming Languages*. ACM Press, 1980.

[53] J.V. Guttag, J.J. Horning, and J.M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[54] Masami Hagiya. Explanation-based generalization in the logical framework. Unpublished manuscript, July 1989.

[55] Masami Hagiya. Generalization from partial parameterization in higher-order type theory. *Theoretical Computer Science*, 63:113–139, 1989.

[56] Masami Hagiya. Programming by example and proving by example using higher-order unification. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 588–602. Springer-Verlag, 1990.

[57] Rogers P. Hall. Computation approaches to analogical reasoning: A comparative analysis. *Artificial Intelligence*, 1988.

[58] John Hannan and Dale Miller. Enriching a meta-language with higher-order features. In John Lloyd, editor, *Proceedings of the Workshop on Meta-Programming in Logic Programming*, Bristol, England, June 1988. University of Bristol.

[59] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 942–959, Cambridge, Massachusetts, August 1988. MIT Press.

[60] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE, June 1987. An extended and revised version is available as Technical Report CMU-CS-89-173, School of Computer Science, Carnegie Mellon University.

[61] Nevin Heintze, Spiro Michaylov, Peter Stuckey, and Roland Yap. On meta-programming in CLP($\Re$). In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 52–66. MIT Press, October 1989.

[62] Walter L. Hill. Machine learning for software reuse. In *Proceedings of IJCAI*, pages 338–344, 1987.

[63] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986. London Mathematical Society Student Texts: 1.

[64] Haym Hirsh. Explanation-based generalization in a logic-programming environment. In *Proceedings of IJCAI*, pages 221–227, 1987.

[65] Haym Hirsh. *Incremental Version-Space Merging: A General Framework for Concept Learning.* PhD thesis, Stanford University, Stanford, CA, June 1989.

[66] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[67] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[68] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[69] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co., Ltd., London, 1968.

[70] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 111–119. ACM Press, January 1987.

[71] Smadar T. Kedar-Cabelli and L. Thorne McCarty. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 383–389, 1987. Also available as tech. report ML–TR–10, Department of Computer Science, Rutgers University.

[72] Richard Keller. Deciding what to learn. Technical Report ML–TR–6, Rutgers University, New Brunswick, NJ, April 1986.

[73] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[74] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Towards chunking as a general learning mechanism. In *Proceedings of AAAI*, 1984.

[75] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[76] Jean-Louis Lassez, Michael J. Maher, and Kimbal G. Marriott. Unification revisited. Technical Report RC 12394, IBM – T.J. Watson Research Center, Yorktown Heights, NY, December 1986.

[77] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25–34. ACM Press, November 1988.

[78] B.P. Lientz and E.F. Swanson. *Software Maintenance Management.* Addison-Wesley, 1980.

[79] Timothy G. Lindhold and Richard A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In Jean-Louis Lassez, editor, *Proceedings of the International Conference on Logic Programming*, pages 21–39. MIT Press, 1987.

[80] Peter Madden. The specialization and transformation of constructive existence proofs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 413–418. IJCAI, 1989.

[81] Sridhar Mahadevan. An apprentice-based approach to learning problem-solving knowledge. Technical Report ML–TR–30, Rutgers University, New Brunswick, New Jersey, May 1990. PhD Dissertation.

[82] Bertrand Meyer. On formalism in specification. *IEEE Software*, pages 6–26, January 1985.

[83] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6(1-2):57–77, January 1989.

[84] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming.* Springer-Verlag LNCS, 1990.

[85] Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Anual Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.

[86] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1989. Available as Ergo Report 88–055, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[87] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*, pages 98–105. IEEE, June 1987.

[88] Dale A. Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Symposium on Logic Programming, San Francisco.* IEEE, September 1987.

[89] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[91] Steven Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of AAAI*, pages 564–569, 1988.

[92] Steven Minton, Jaime Carbonell, Craig Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40(1–3):63–118, 1989. Special issue on machine learning. Also available as technical report CMU–CS–89–103, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[93] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, March 1982.

[94] Tom M. Mitchell. Learning and problem solving. In *Proceedings of IJCAI*, pages 1139–1151. Morgan Kaufmann, 1983.

[95] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[96] Tom M. Mitchell, Sridhar Mahadevan, and Louis Steinberg. LEAP: A learning apprentice for VLSI design. In *Proceedings of IJCAI*, pages 573–580. Morgan Kaufmann, 1985.

[97] Jack Mostow. Why are design derivations hard to replay. In Tom M. Mitchell, Jaime G. Carbonell, and Ryszard S. Michalski, editors, *Machine Learning: A Guide to Current Research.* Kluwer Academic, 1986.

[98] Jack Mostow. Design by derivational analogy: Issues in the automated replay of design plans. *Artificial Intelligence*, 40(1–3):119–184, 1989. Special issue on machine learning.

[99] Jack Mostow and T. Fawcett. Approximating intractable theories: A problem space model. Technical Report ML–TR–16, Rutgers University, New Brunswick, NJ, December 1987.

[100] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[101] Mogens Nielson, Klaus Haveland, Kim Ritter Wagner, and Chris George. The RAISE language, methods and tools. *Formal Aspects of Computing*, 1989(1):85–114, 1989.

[102] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, 1983.

[103] Fernando C.N. Pereira. Prolog and natural-language analysis: Into the third decade. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 813–832. MIT Press, 1990.

[104] Frank Pfenning. Program development through proof transformation. In Wilfried Sieg, editor, *Logic and Computation*, Contemporary Mathematics. AMS, Providence, Rhode Island, 1988. Available as Ergo Report 88–047, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[105] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society Press, July 1991.

[106] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988.

[107] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[108] Armand E. Prieditis and Jack Mostow. Prolearn: Toward a Prolog interpreter that learns. In *Proceedings of AAAI*, Spring 1987.

[109] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[110] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, New York, 1972. ACM.

[111] C. Rich. A formal representation for plans in the programmer's apprentice. In *Proceedings of IJCAI*, pages 1044–1052, 1981.

[112] Paul S. Rosenbloom and John E. Laird. Mapping explanation-based generalization onto Soar. In *Proceedings of AAAI*, pages 561–567, 1986.

[113] Neil C. Rowe. *Artificial Intelligence Through Prolog.* Prentice-Hall, 1988.

[114] Stuart J. Russell and Benjamin N. Grosof. A sketch of autonomous learning using declarative bias. In Paul B. Brazdil and Kurt Konolige, editors, *Machine Learning, Meta-Reasoning and Logics*, pages 18–53. Kluwer Academic, 1990.

[115] William L. Scherlis. Program improvement by internal specialization. In *Eighth Symposium on Principles of Programming Languages*, pages 41–49. ACM, ACM, January 1981.

[116] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R.E.A. Mason, editor, *Information Processing*, pages 199–212. Elsevier Science Publishers, 1983.

[117] Ehud Y. Shapiro. Logic program with uncertainties: A tool for implementing rule-based systems. In *Proceedings of IJCAI*, pages 529–532, 1983.

[118] Jude Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5(1):39–70, March 1990.

[119] M. Sintzoff. Suggestions for composing and specifying program design decisions. In *International Symposium on Programming.* Springer-Verlag, 1980. Lecture Notes in Computer Science.

[120] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

[121] Douglas R. Smith and Thomas T. Pressburger. Knowledge-based software development tools. Technical Report KES.U.87.6, Kestrel Institute, June 1987.

[122] J.M. Spivey. *The Z Notation.* Prentice-Hall, 1989.

[123] David Steier. *Automating Algorithm Design within a General Architecture for Intelligence.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1989. Available as CMU-CS-89-128.

[124] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[125] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press Series in Computer Science. MIT Press, 1977.

[126] Prasad Tadepalli. Learning in intractable domains. In Tom M. Mitchell, Jaime G. Carbonell, and Ryszard S. Michalski, editors, *Machine Learning: A guide to Current Research*, pages 337–341. Kluwer Academic, 1986.

[127] Akikazu Takeuchi and Foichi Furukawa. Partial evaluation of Prolog programs and its application to meta programming. In H.J. Kugler, editor, *Information Processing*, pages 415–420. Elsevier Science Publishers, 1986.

[128] P.B. Thistlewaite, M.A. McRobbie, and R.K. Meyer. *Automated Theorem-Proving in Non-Classical Logics.* Pitman, London, 1988.

[129] Frank van Harmelen and Alan Bundy. Explanation-based generalization = partial evaluation. *Artificial Intelligence*, 36:401–412, 1988.

[130] Steven A. Vere. Induction of concepts in the predicate calculus. In *IJCAI*, pages 281–287, 1975.

[131] Lincoln A. Wallen. *Automated Proof Search in Non-Classical Logics: Efficient Matrix Proof Methods for Modal and Intuitionistic Logics.* PhD thesis, University of Edinburgh, 1987.

[132] D.H.D. Warren. An abstract Prolog instruction set. Technical Report 309, Artificial Intelligence Center, SRI International, October 1983.

[133] Richard Waters. The programmer's apprentice: A session with kbemacs. *IEEE Transactions on Software Engineering*, SE-11(11):1296–1320, November 1985.

[134] David S. Wile. Local formalisms: Widening the spectrum of wide-spectrum languages. In L.G.L.T. Meertens, editor, *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Toelz, FRG*, pages 459–481. North-Holland, November 1986.

[135] Jeannette M. Wing. A specifier's introduction of formal methods. *Computer*, 23(9):8–24, September 1990. (Special issue on formal methods).